Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Розробка платформи для випуску, продажу та обліку сертифікатів енергетичних атрибутів, реалізованих за допомогою розширеної UTXO моделі токена. Імплементація Hyperledger Fabric / Fiware Orion Context Broker адаптера.

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
ас.
Гороховський К.С.
_____ (Підпис)
"___" _____ 2022 року
Виконав студент
КН-4 Холодов Д. В.
"___" _____ 2022 року

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

ас.

_____ Гороховський К.С.

„_____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Холодову Дмитру Віталійовичу

факультету інформатики 4 курсу бакалаврської програми

ТЕМА: **Розробка платформи для випуску, продажу та обліку сертифікатів енергетичних атрибутів, реалізованих за допомогою розширеної UTXO моделі токена. Імплементація Hyperledger Fabric / Fiware Orion Context Broker адаптера.**

Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Вступ
Розділ 1. Дослідження та аналіз предметної області
Розділ 2. Розділ 2. Аналіз технічного завдання
Розділ 3. Розробка застосунку
Висновки
Список літератури
Додатки (за необхідністю)

Дата видачі „___” _____ 2021 р.

Керівник_____

(підпис)

Завдання отримав_____

(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	01.10.2021	
2.	Огляд літератури за темою роботи	01.11.2021	
3.	Проведення дослідження	01.12.2021	
4.	Написання програмного застосунку	01.01.2022	
5.	Написання текстової частини	04.04.2022	
6.	Захист курсової роботи	20.05.2022	

Студент _____

Керівник _____ “_____”_____ 2022

Зміст

Перелік умовних позначень:

EAC - сертифікат енергетичних атрибутів.

Токен – інструмент для передачі права власності в цифровому вигляді.

UTXO - вихід невитрачених транзакцій, кількість токенів, що залишилася після виконання транзакції.

Смарт-контракт - програмне забезпечення, яке описує бізнес-логіку створення, оновлення та видалення даних в мережі блокчейн.

DLT - технологія розподіленого реєстру.

Анотація

Курсова робота присвячена проблемам збору даних про “зелену” електроенергію та гарантіям походження цих даних у вигляді сертифікатів енергетичних атрибутів (EAC). За метод вирішення проблеми було обрано реалізацію розширеної UTXO моделі токена для фіксації гарантії походження чистої електроенергії у вигляді токена в блокчейні.

Розроблено програмний застосунок для випуску, продажу та обліку сертифікатів енергетичних атрибутів (EACs), реалізованих за допомогою розширеної UTXO моделі токена. Реалізовано DLT-адаптер для фреймворку Hyperledger Fabric та Fiware Orion Context Broker як спосіб вирішення задачі збору даних про видобуту електроенергію.

Досліджено UTXO модель токена як підхід до вирішення задачі гарантії походження даних.

Текстова частина курсової містить опис дослідження проблем збору та гарантій походження даних про екологічно чисто видобуту електроенергію та спосіб їх вирішення. Текстова частина курсової також описує необхідні теоретичні відомості про використані інструменти та технології.

Ключові слова: блокчейн, UTXO, EAC, Fiware, Hyperledger Fabric, чиста енергетика, Nest.js

Вступ

Актуальність теми

Забезпечення здорового способу життя та сприяння благополуччю для всіх у будь-якому віці – важливі складові сталого розвитку. ООН запропонувала 17 цілей для трансформації нашого світу. Цілі у сфері сталого розвитку є своєрідним закликом до дії, що походить від усіх країн — бідних, багатих та середньорозвинених [1]. Ціль №7 це забезпечення загального доступу до недорогих, надійних, стійких та сучасних джерел енергії для всіх. Енергетика є домінуючим фактором у галузі зміни клімату і на її частку припадає **близько 60 відсотків** загального обсягу глобальних викидів парникових газів.

Ціль №7 полягає у забезпеченні загального доступу до недорогих, надійних, стійких та сучасних джерел енергії для всіх. Це важливо тому, що наше повсякденне життя залежить від надійних і недорогих енергетичних послуг, а також від їх безперебійності та рівноправного розвитку. Енергетична система влаштована належним чином, сприяє розвитку всіх секторів: від підприємництва, охорони здоров'я та освіти до сільської господарства, інфраструктури, комунікацій та високих технологій. І навпаки, відсутність доступу до енергопостачання та систем перетворення енергії представляє собою перешкоду для розвитку людського потенціалу та економіки.

Ця ціль важлива навіть для тих хто має доступ до електрики, оскільки протягом багатьох десятиліть основними джерелами електроенергії були такі види викопного палива, як вугілля, нафта і газ, проте при спалюванні вуглеводневого палива відбувається викид в атмосферу парникових газів, які викликають зміну клімату та негативно впливають на благополуччя людей та довкілля. Від цього страждає все населення планети, а не окрема його частина. Крім того, у всьому світі стрімко зростає рівень споживання електроенергії. Отже, за відсутності

стабільного електропостачання країни не зможуть підживлювати свою економіку.

Що ми можемо зробити? Країни можуть прискорити перехід до недорогої, надійної та стійкої енергетичної системи, інвестуючи в відновлювані джерела енергії, віддаючи пріоритет енергоефективним практикам, а також використовуючи технології та інфраструктуру, засновані на чистій енергії. Підприємства можуть підтримувати та охороняти екосистеми з метою використання та подальшого розвитку гідроенергетики та біоенергетики, а також взяти на себе зобов'язання із забезпечення всіх 100 відсотків необхідної їм енергії тільки з поновлюваних джерел. Інвестори можуть збільшити обсяг вкладень у послуги з надання стійкої енергії, швидко виводячи на ринок нові технології, що надаються різноманітною базою постачальників [2].

Об’єкт дослідження

Гарантія походження чистої електроенергії.

Предмет дослідження

UTXO модель токена як спосіб гарантування походження чистої електроенергії.

Мета дослідження

Визначити чи можливе використання розширеного UTXO токена як гарантії походження даних про виробництво чистої електроенергії.

Постановка задачі

1. Аналіз засобів для забезпечення гарантії походження сертифікатів енергетичних атрибутів.
2. Дослідження моделей токена.
3. Дослідження засобів збору даних з IoT пристроїв та вибір найбільш актуального.
4. Розробка власної блокчейн системи та бекенд сервісу для взаємодії з нею, основна мета якої – токенізація сертифікатів енергетичних атрибутів з метою гарантії їх походження та їх продаж.
5. Розробка смарт-контракту обраної моделі токена.

Структура роботи:

1) В першому розділі викладено загальні відомості про предметну область, описано наявні проблеми та запропоновано, та обґрунтовано підхід до їх вирішення. Крім цього, проаналізовано існуючі рішення на ринку.
2) В другому розділі проаналізовано технічне завдання: описано користувачів, функціональні вимоги та вимоги до даних.
3) В третьому розділі обґрунтовано вибір інструментів, детально описано реалізацію ключових компонентів системи та наведено результати роботи програми.

Розділ 1. Дослідження та аналіз предметної області

1.1 Загальні відомості

Сертифікати енергетичних атрибутів (EAC) видаються як доказ електроенергії, виробленої з відновлюваних джерел. Кожен EAC стверджує, що електроенергію було вироблено та введено в мережу певним відновлюваним джерелом, таким як вітрова або сонячна електростанція. Зокрема EAC є інструментом що забезпечує прозорість та достовірність процесу трансформації нашого світу згідно з 17 цілями ООН.

У 2021 році було видано 835 мільйонів сертифікатів (825 ТВт-год). Ринок “зелених” сертифікатів продовжує зростати [3]. Сертифікати часто торгуються за цінами, що визначаються ринком попиту та пропозиції. Коли відповідний виробник енергії виробляє електроенергію, він отримує сертифікати на відповідний обсяг виробленої енергії, який може зберігатися виробником, випускатися на ринок або передаватися третім сторонам, таким як кінцеві споживачі. Їх можна продавати як у комплекті з електроенергією, так і окремо. Наприклад, компанія яка використала впродовж року 100 МВт електроенергії із загальної мережі може придбати EAC на 100 МВт і в результаті прозвітуватися що вона є 100% вуглецево-нейтральною, а в майбутньому уникнути “зелених” податків (податків на вуглець). Тобто, ці сертифікати стали своєрідною валютою, ліквідним активом на ринку відновлюваної енергії, допомагаючи покупцям достовірно заявити про свій **вибір стійкої енергії** та зменшити вплив на навколишнє середовище.

Покупці роблять внесок у винагороду компаніям за електроенергію, вироблену з відновлюваних джерел, купуючи сертифікати, тим самим підтверджуючи прихильність покупців до захисту навколишнього середовища [2].

1.2 Проблематика

У країнах ЄС діє законодавство про гарантії походження електроенергії та «зелені» сертифікати, які можуть бути продані або використані для доказу того, що споживана електроенергія дійсно була згенерована з відновлюваних джерел [4].

Торгівля ЗС здійснюється на енергетичній біржі, що потребує створення окремого паралельно функціонуючого ринку для здійснення обігу сертифікатів, не прив’язаного до руху електроенергії, на основі якої вони випускаються [5].

В цьому і полягають основні труднощі для виробників відновлюваної енергії. Незалежні аудитори повинні сертифікувати електроенергію як "зелену". Тоді виробник може продати сертифікати енергетичних атрибутів клієнтам, які хочуть використовувати чисту енергію. Бо в електромережі неможливо відокремити зелену електроенергію від звичайної. Таким чином, лише сертифікат може довести, що ваша компанія використовує відновлювані джерела енергії. Тобто стоїть нетривіальна задача гарантувати походження електроенергії без необхідності залучення 3ї сторони [6].

1.3 Аналіз наявних рішень поставленої задачі

Blockchain for Climate Foundation створила платформу BITMO, щоб забезпечити випуск та обмін «Blockchain International Transfered Mitigation Outcomes» (BITMOs) як ERC-1155 Non-Fungible Tokens (NFT) на блокчейні Ethereum. Кожен токен, еквівалентний одній тонні еквіваленту CO2, має всі відповідні дані про вуглецевий кредит, вбудований прямо в NFT. Ця платформа може виконувати важливу роль у

інфраструктурі, забезпечуючи відповідні коригування та негайне врегулювання торгів за національними запасами вуглецю. Результати скорочення викидів є найважливішими і незабаром стануть одними з найцінніших активів світу.

Інші знайдені наявні рішення на ринку мають одну спільну характеристику, вони є централізованими, внаслідок чого виникає необхідність в послугах третіх сторін, аудиторів.

Fiware Orion Context Broker є відомим в ЄС відкритим рішенням, але наразі не має DLT адаптеру для роботи з фреймворком Hyperledger Fabric.

1.4 Опис обраного підходу до вирішення поставленої задачі

Глобальну зміну клімату можна подолати лише за допомогою міжнародної співпраці. Добре функціонуючі, ефективні, надійні енергетичні ринки дозволяють капіталу текти туди, де його можна найкраще використати для зменшення викидів і покращення життя.

Торгові платформи на базі блокчейну мають на меті скоротити транзакційні витрати, підвищити конкурентність і відкрити доступ для малих і середніх виробників, які можуть продавати енергію іншим учасникам мережі.

Фактично ринком стає безпосередньо середовище блокчейну, де відсутній посередник, а споживачі купують необхідну кількість енергоресурсу безпосередньо у виробника за прописаною в смарт-контракті бізнес-логікою. Сам енергоресурс зберігається в мережі у вигляді розширеного UTXO токена. Інформація щодо того на якій станції була видобута електроенергія, якого вона типу (сонячна, вітрова, ..), за

який проміжок часу добута та в якій кількості - збиратиметься за допомогою IoT вимірювальних пристроїв та контекстного брокеру Fiware Orion, замість звичного та неефективного, з точки зору швидкості та точності, підходу з використанням живої сили.

Прозорість підходу сприяє підвищенню рівня довіри між учасниками процесу, чого сьогодні не вистачає для конкурентного функціонування вітчизняних енергетичних ринків та утворення справедливої ціни для українських та європейських споживачів.

Управління проєктами та співфінансування на базі блокчейну реалізується за рахунок прозорості операцій, низьких затрат на проведення транзакцій та утримання такої системи. [4]

1.5 Обґрунтування вибору рішення

Технологія блокчейн дозволяє уникати подвійних розрахунків і витрат на послуги аудиторів за рахунок:

- прозорості транзакцій з випуску та торгівлі «зелених» сертифікатів в мережі
- стійкості реєстру сертифікатів (UTXO токенів) до втручання, імутабельності даних
- спільних для всіх учасників мережі алгоритмів, правил створення та змінення активів (енергетичних сертифікатів) в мережі

Існують дві найбільш популярні моделі токена: UTXO модель, найвідомішим представником якої є Bitcoin, та Account/Balance модель, найвідомішим представником якої є токен Ether в мережі Ethereum.

Кожен блокчейн, незалежно від того, використовує він модель UTXO чи Account/Balance, дотримується цієї схеми. Взаємодія користувача, переважно транзакції, транслюються в мережу, і з кожним новим блоком їх набір постійно записується. Баланси сторін угоди оновлюються, коли система переходить у новий стан. Різниця між UTXO та Account/Balance моделями полягає в тому, як ведеться бухгалтерський облік. Під бухгалтерським обліком мається на увазі запис стану та перехід з одного стану в інший.

Перша істотна відмінність між двома моделями балансу полягає в тому, як реєструється стан системи. У моделі UTXO переміщення активів записується у вигляді орієнтованого ациклічного графу між адресами, тоді як модель облікового запису підтримує базу даних станів мережі.

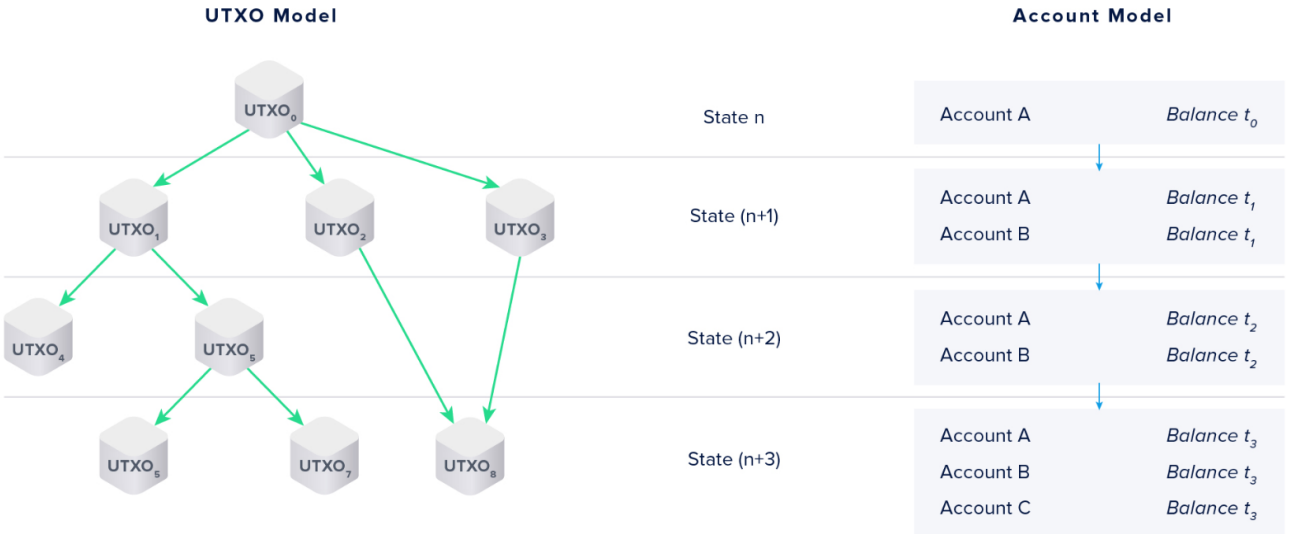


Рис 1.1 - запис станів системи [7]

Граф визначається як набір вузлів або вершин, з'єднаних ребрами. У орієнтованому графі кожне ребро має напрямок, зазвичай вказаний стрілками. Орієнтовані ациклічні графи не допускають циклічних відносин між вузлами. На графіку вище зліва показано орієнтований ациклічний граф моделі UTXO. Кожен стан представляє блок в ланцюзі блоків. Кожен вихід транзакції є вузлом у графі, і кожна транзакція представлена одним або кількома ребрами, що походять із виходу транзакції. Отже, вихідні дані невитраченої транзакції не мають вихідного ребра. У наведеному вище прикладі виходи транзакції 4, 5, 7 і 8 є невитраченими. Праворуч на графіку показано представлення різних станів у моделі облікового запису (Account/Balance). З кожним новим блоком стан системи оновлюється відповідно до транзакцій, що містяться в блоці. Кількість облікових записів залишається постійною і не залежить від кількості проведених транзакцій, доки кількість користувачів або смарт-контрактів залишається незмінною.

У моделі UTXO весь граф виходів транзакцій, витрачених і невитрачених, представляє глобальний стан. В Account/Balance моделі лише поточний набір рахунків та їх залишки представляють глобальний стан. У наведеному вище прикладі це набір рахунків (акаунтів) A, B і C та їх відповідні залишки.

Концептуальна відмінність полягає в тому, що Account/Balance модель оновлює баланси користувачів глобально в системі. Модель UTXO натомість записує лише квитанції або результати транзакцій. У моделі UTXO залишки на рахунках розраховуються на стороні клієнта шляхом додавання доступних невитрачених результатів транзакцій [7].

1.5.1 Обгрунтування вибору UTXO моделі

Транзакції UTXO визначають результуючий стан, але в моделі облікового запису результуючий стан залежить від попереднього. Тому, під час виконання транзакцій у системі на основі рахунків кожна транзакція повинна оброблятися послідовно, коли йдеться про одну й ту саму адресу. Однак модель UTXO може обробляти свої транзакції паралельно, оскільки неможливо, щоб дві транзакції вплинули на один і той же UTXO.

Для систем на основі облікових записів необхідно вжити заходів обережності, щоб гарантувати, що раніше підписані транзакції не відтворюються в мережі. Зазвичай це відбувається у формі одноразового числа (nonce), що збільшується, забезпечуючи унікальність. Для систем на основі UTXO це не проблема, оскільки кожен UTXO споживається і не може бути використаний знову.

Хоча, модель облікового запису спрощує реалізації UI/UX з точки зору відображення балансів для конкретних адрес, коли відображення цієї інформації в системі на основі UTXO трохи складніше. Оскільки більшість користувачів знайомі з рахунками та балансами, це потрібно реалізувати шляхом підсумовування UTXO заданих адрес.

З іншої точки зору, розбиття або шардинг блокчейн мережі на сегменти або сайдчейни також простіше при використанні моделі UTXO. Агрегування витратних результатів транзакцій і визначення результатів відбувається на стороні клієнта, зменшуючи навантаження на загальну систему. У моделі облікового запису кожна нода повинна локалізувати обліковий запис відправника і одержувача в різних сегментах і редагувати обидва.

Тобто як правило, модель UTXO дозволяє розбити дані простіше.

Отже, не дивлячись на певні особливості при роботі з UTXO моделлю, вона дозволяє обробляти транзакції паралельно, а отже швидше, має вбудований захист від подвійних витрат та дозволяє розділити сховище даних на кілька сегментів, щоб підтримувати продуктивність кожного окремого розділу, в свою чергу, покращуючи продуктивність всієї системи [7].

1.5.2 Обгрунтування вибору Fiware Orion Context Broker

Сьогодні більшість навіть компаній, навіть компанії-гіганти енергетичного сектору використовують живу силу для збору даних. Тобто людина вручну знімає показники з лічильників електростанцій/генераторів після чого записує їх в таблицю Excel. Такий підхід є досить повільним та неточним через людський фактор. Саме тому

пропонується використання IoT зчитувальних пристроїв. IoT девайси можуть комунікувати за допомогою різних протоколів передачі даних. Fiware Orion Context Broker для вирішення проблеми збору даних з пристроїв що використовують різні протоколи пропонує використання компоненту фреймворку – IoT Agent.

Агент IoT — це компонент, який дозволяє групі пристроїв надсилати свої дані до контекстного брокеру і керувати ними за допомогою своїх власних протоколів. Кожен агент IoT визначено для одного формату корисного навантаження, хоча вони можуть використовувати та обробляти кілька різнорідних форматів цього корисного навантаження [8].

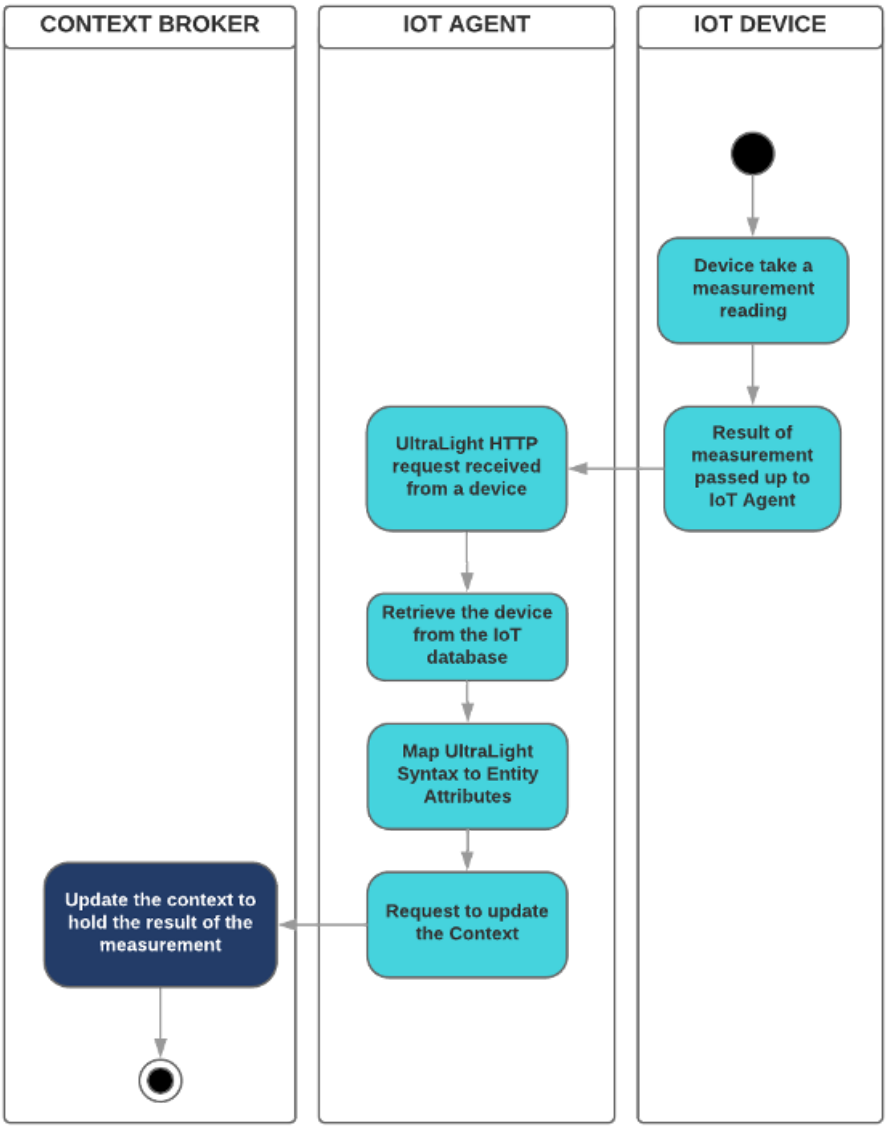


Рис 1.2 - діаграма активності роботи контекстного брокеру з IoT приладами [8]

Розділ 2. Аналіз технічного завдання

Головна мета застосунку

Гарантувати походження чистої електроенергії шляхом випуску UTXO токенів в блокчейні та дозволити користувачам торгувати сертифікатами енергетичних атрибутів записаними в UTXO токен.

2.1 Опис користувачів в системі

В системі наявні наступні ролі користувачів - інвестор та власник компанії.

2.2 Технічні вимоги користувачів до функціональності системи

Власник компанії повинен мати можливість:

- створити компанію
- передивлятися свої компанії
- створити електростанцію
- передивлятися свої електростанції
- передивлятися показники електростанції
- випустити сертифікат енергетичних атрибутів у вигляді UTXO токена в блокчейні
- передивлятися власні токени в мережі
- виставити сертифікат на продаж
- передивлятися виставлені сертифікати на продаж
- поповнити баланс фіатного гаманця
- передивлятися баланс фіатного гаманця

Інвестор повинен мати можливість:

- передивлятися виставлені сертифікати на продаж
- придбати сертифікат енергетичних атрибутів
- поповнити баланс фіатного гаманця
- передивлятися баланс фіатного гаманця

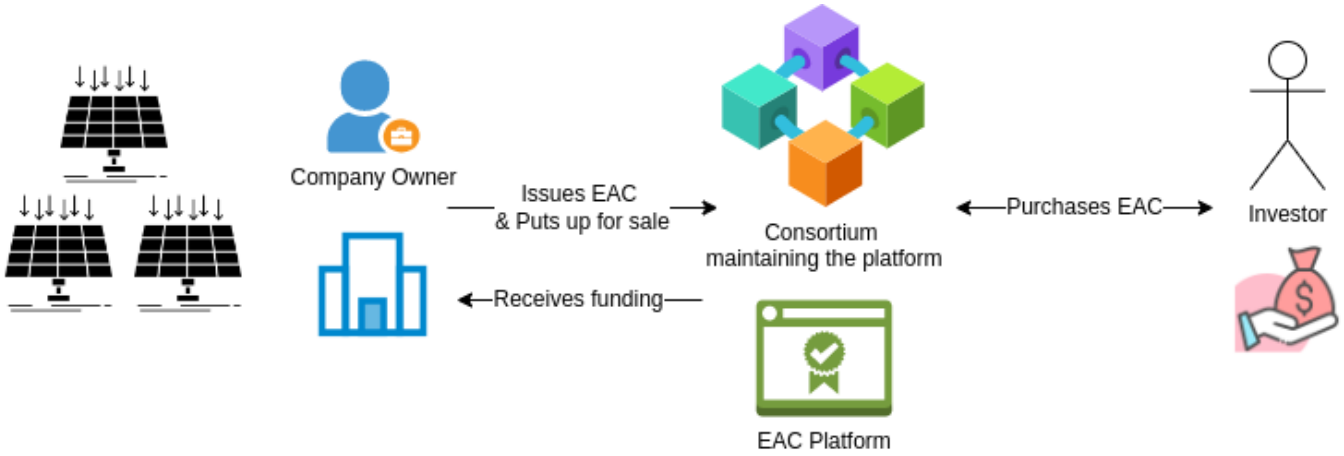


Рис 2.1 - Функціональна схема взаємодії

Учасниками консорціуму можуть бути різні організації та інститути з енергетичного сектору різних країн.

2.3 Специфікації вимог до даних

Про користувачів системи мають зберігатися наступні дані: унікальний ідентифікатор, повне ім’я, унікальна електронна адреса, роль, пароль.

Про організацію мають зберігатися наступні дані: реєстраційний номер, назва, ID власника, тип компанії (приватна, публічна), адреса організації, електронна адреса організації.

Про електростанцію мають зберігатися наступні дані: реєстраційний номер організації, назва, тип станції (сонячна, вітрова), продуктивність (потужність), дата виробництва та введення в експлуатацію, країна та регіон в якій розташована, країна виробник.

Про сертифікат енергетичних атрибутів мають зберігатися наступні дані дата початку та кінця виробництва, обсяг згенерованої електроенергії, ID електростанції, країна та регіон в яких розташована станція, тип станції, країна виробник, дата виробництва, дата введення в експлуатацію, потужність станції

Розділ 3. Розробка застосунку

3.1 Обґрунтування вибору засобів розробки

3.1.1 Fabric

Для розгортання власної мережі блокчейн було обрано фреймворк Hyperledger Fabric v2.2. Це фреймворк для побудови enterprise-level блокчейнів. Дозволяє побудувати архітектуру блокчейн мережі таким чином, що учасники цієї мережі можуть досягти певного консенсусу, певного ступеня довіри між собою.

Він дозволяє гнучко налаштовувати приватну (permissioned) мережу, де на відміну від загальнодоступної мережі, учасники відомі один одному, а не анонімні, тобто, повністю недовірені. Це означає, що хоча учасники можуть не довіряти повністю один одному (наприклад, вони можуть бути конкурентами в одній галузі), мережа може працювати за моделлю управління, яка побудована на основі довіри між учасниками, наприклад юридична угода або основа для вирішення спорів. У такому дозволеному контексті ризик навмисного введення учасником шкідливого коду за допомогою смарт-контракту зменшується. Замість того, щоб бути повністю анонімним, винну сторону можна легко ідентифікувати та обробити інцидент відповідно до умов моделі управління.

Fabric перейняв традиційну модель інфраструктури відкритих ключів (PKI): кожен з учасників мережі — активні елементи всередині або поза мережею, здатні користуватися послугами — має цифрову особу, інкапсульовану у цифровий сертифікат X.509. Цифровий сертифікат — це документ, який містить набір атрибутів, що стосуються власника сертифіката. Ці ідентифікатори мають дуже важливе значення, оскільки вони визначають точні права на ресурси та доступ до інформації, які мають суб’єкти в мережі блокчейн.

За випуск сертифікатів відповідє сертифікаційний центр (Caertificate Authority).

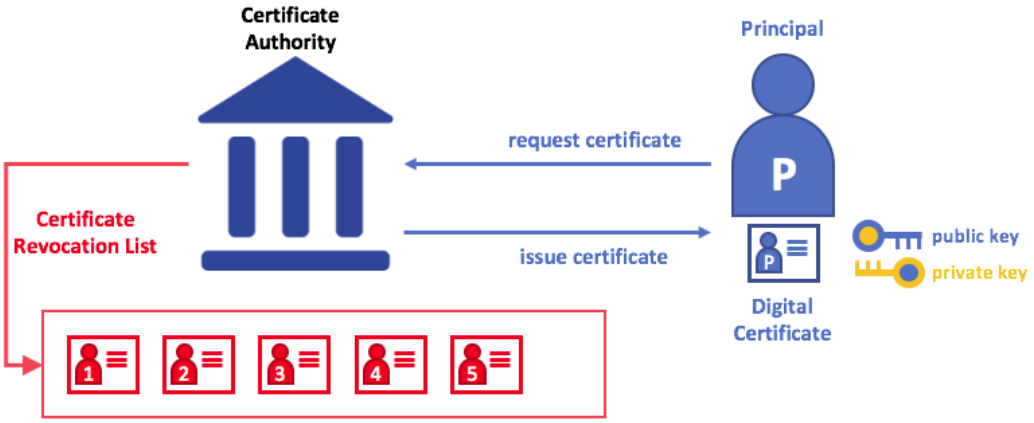


Рис 2.2 - сертифікаційний центр [9]

Окрім цього, модульна архітектура фреймворку дозволяє розгорнути нову мережу та підключити її до вже існуючого застосунку. Вона також надає можливість створювати надбудови над фреймворком, такі як власна система управління ідентифікацією користувачів. Особливо корисною перевагою такої архітектура фреймворку для мене була можливість створювати та оновлювати чейнкод (chaincode) в каналі, без необхідності заново розгортати всю блокчейн мережу. В enterprise level застосунку також корисними будуть можливість гнучко налаштовувати політики, навіть після запуску мережі, а також використовувати різні імплементації алгоритмів відмовостійкості (pluggable consensus).

3.1.2 Fiware

Фреймворк Fiware було обрано як відкриту стійку екосистему безкоштовних інструментів, які спрощують розробку та використання нових розумних додатків і бізнес-моделей у багатьох секторах. З практичної точки зору він дозволяє легко інтегрувати в свою систему роботу з IoT приладами, що працюють на різних протоколах передачі даних [10].

3.1.3 Nest.js

Nest — це платформа для створення ефективних, масштабованих серверних додатків на Node.js. Він повністю підтримує мову TypeScript, яка є строго типізованою, що дозволяє на ранніх етапах знаходити помилки, і поєднує елементи OOP (об’єктно-орієнтоване програмування), FP (функціональне програмування) і FRP (функціональне реактивне програмування).

Під капотом Nest використовує надійні фреймворки HTTP-сервера, такі як Express (за замовчуванням), і за бажанням його можна також налаштувати на використання більш легкого Fastify.

Nest забезпечує рівень абстракції над поширеними фреймворками Node.js (Express/Fastify), але також надає їхні API безпосередньо розробнику. Це дає розробникам свободу використовувати безліч сторонніх модулів, які доступні для базової платформи [11].

3.1.4 Docker

Docker – це платформа для розробки, доставки та запуску контейнерних програм. В проєкті використана для швидкого та зручного розгортання блокчейн мережі та контекстного брокеру.

3.2 Структура баз даних

В застосунку використано 3 бази даних. Fiware Orion Context Broker використовує MongoDB за замовчуванням. На бекенді використано PostgreSQL для зберігання ключових сутностей в системі. Використано саме SQL базу даних для забезпечення цілісності даних, ізольованості та атомарності транзакцій. Hyperledger Fabric використовує CouchDB для зберігання реального стану активів (World State). World State є результатом всіх транзакцій в блокчейні. CouchDB надає багатий набір інструментів для написання запитів

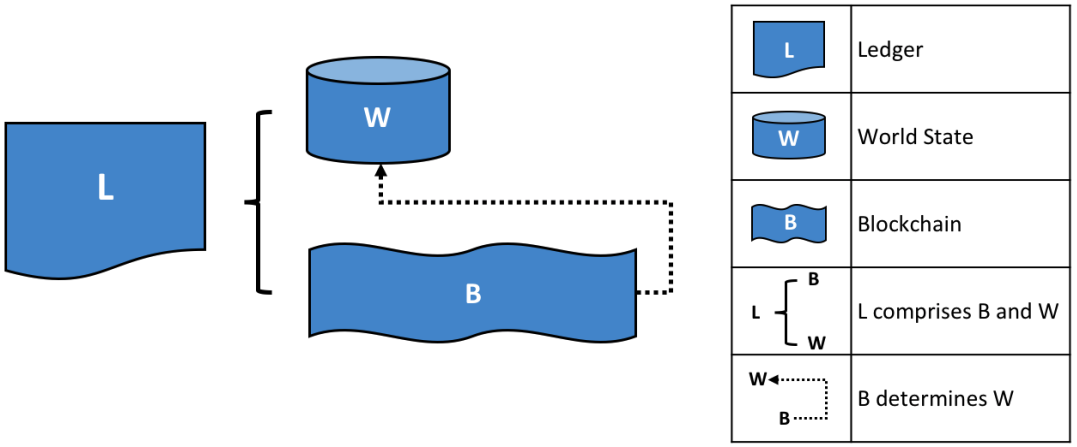


Рис 3.1 - леджер в Hyperledger Fabric [12]

3.2.1 Використання TypeORM

Для інтеграції з базами даних SQL і NoSQL Nest надає пакет @nestjs/typeorm. Nest використовує TypeORM, оскільки це найзріліший об’єктний реляційний маппер (ORM), доступний для TypeScript. Оскільки він написаний на TypeScript, він добре інтегрується з фреймворком Nest. Більш детально використання TypeORM описано в наступних розділах [11].

3.2.2 ЕР-модель

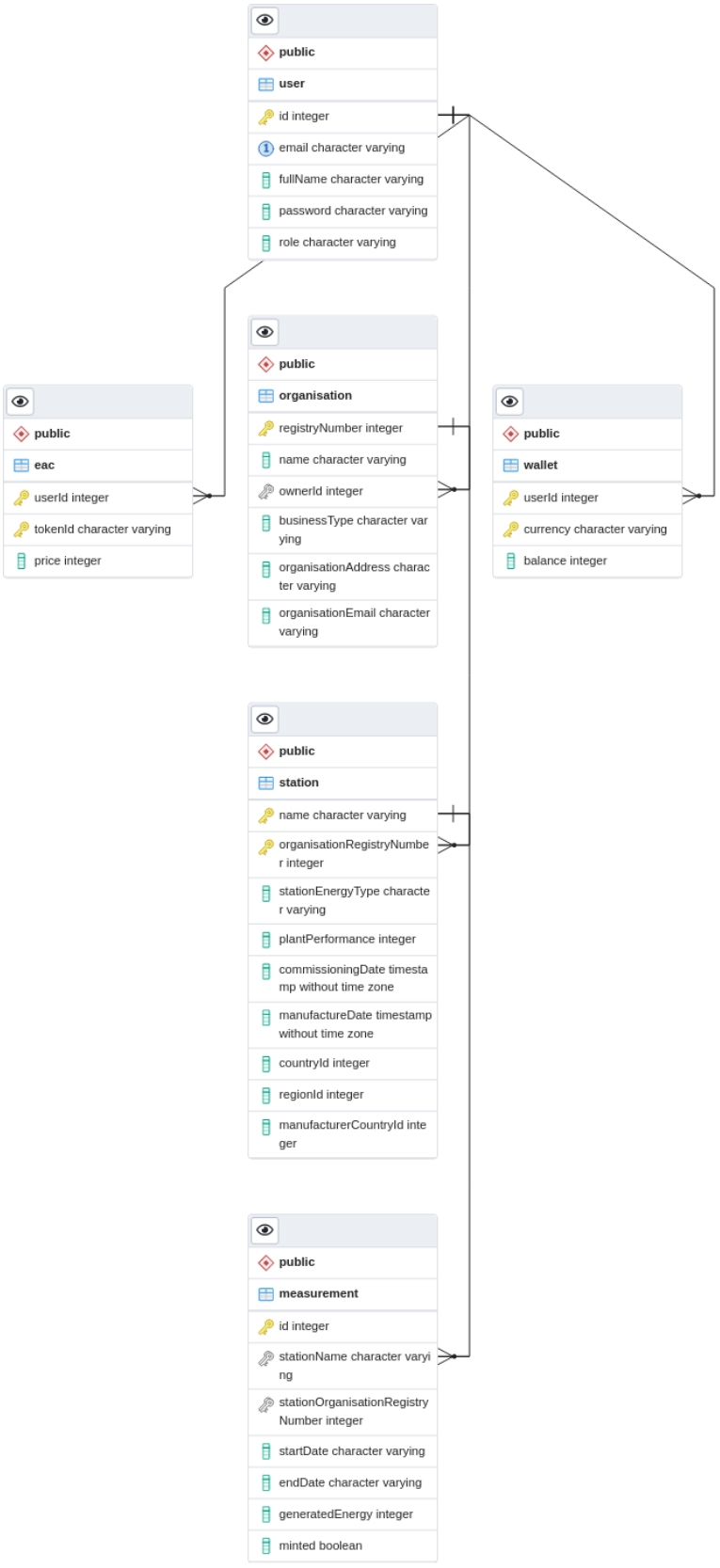


Рис 3.2 - ER-модель

3.3 Проектування архітектури

Для роботи застосунку необхідна взаємодія трьох сервісів: мережі блокчейн з розгорнутим в ній Chaincode, backend сервіс для взаємодії з блокчейн мережею за допомогою API та контекстний брокер який сповіщуватиме backend при появі нових показників виробленої електроенергії. Кожен сервіс використовує свою базу даних в залежності від вимог до даних та вимог до роботи з ними.

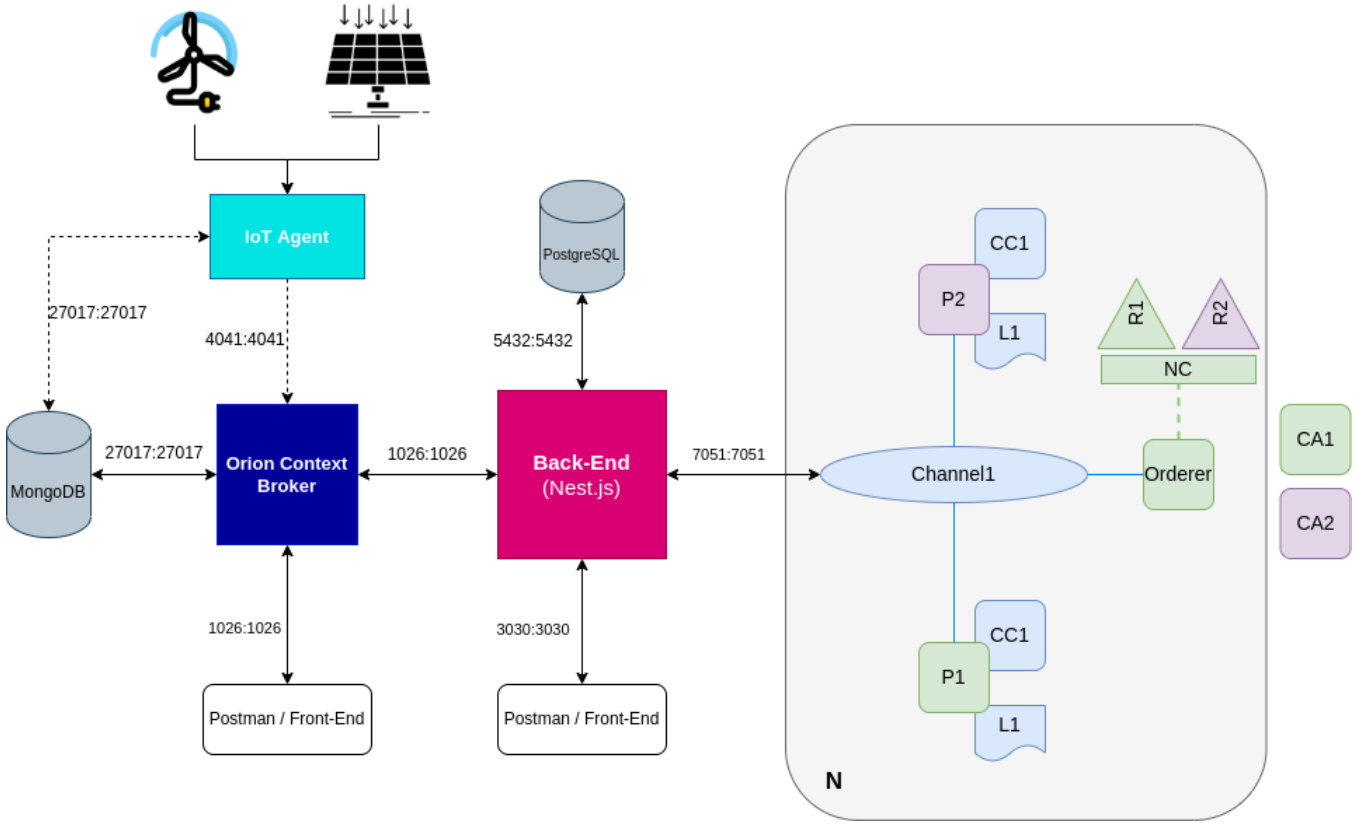


Рис 3.3 - архітектура платформи для випуску та трейдингу EAC

3.4 Реалізація сервісів

3.4.1 Реалізація Chaincode на GoLang

Розширена UTXO модель матиме наступну структуру:

```
type EAC struct {
    ProductionStartDate  string  `json:"prod_start_date"`
    ProductionEndDate  string  `json:"prod_end_date"`
    GeneratedEnergy float64 `json:"generated_energy"`
    StationID string `json:"station_uid"`
    StationLocation string `json:"station_location"`
    StationEnergyType string `json:"station_energy_type"`
    ManufacturerCountryId float64 `json:"manufacturer_country_id"`
    ManufactureDate string `json:"manufacture_date"`
    CommissioningDate string `json:"commissioning_date"`
    PlantPerformance float64 `json:"plant_performance"`
}

// UTXO represents an unspent transaction output
type UTXO struct {
    Key                 string              `json:"utxo_key"`
    Owner               string              `json:"owner"`
    Amount              float64             `json:"amount"`
    EAC                 *EAC                `json:"EAC" metadata:"EAC"`
}
```

Рис 3.4 - структура розширеного UTXO токена

Key, Owner, Amount - набір полів UTXO токена за замовчуванням. Ми ж додатково записуємо в токен дані про згенеровану енергію.

3.4.1.1 Mint

```
// Mint creates a new unspent transaction output (UTXO) owned by the minter
func (s *SmartContract) Mint(ctx contractapi.TransactionContextInterface, amount float64, eac EAC) (*UTXO, error)
```

Функція першим аргументом приймає об’єкт TransactionContext, як і всі функції контракту. Контекст транзакції виконує дві функції. По-перше, дозволяє розробнику визначати та зберігати змінні користувача між викликами транзакцій смарт-контракта. По-друге, він надає доступ до широкого спектру Fabric API, який дозволяє виконувати операції, пов’язані з детальною обробкою транзакцій. Вони варіюються від читання або оновлення леджеру, як незмінного блокчейну, так і змінюваного реального стану (world state), до отримання цифрової особи, що надсилає транзакцію.

Далі отримуємо унікальний ID сертифіката цифрової особи.

```
// Get ID of submitting client identity
minter, err := ctx.GetClientIdentity().GetID()
if err ≠ nil {
    return nil, fmt.Errorf("failed to get client id: %v", err)
}
```

Після чого робимо необхідні перевірки вхідних даних, а саме поля amount.

```
if amount ⩽ 0 {
    return nil, fmt.Errorf("mint amount must be a positive integer")
}
```

Формуємо сам UTXO токен, унікальний ключ токена генерується отримуючи ID транзакції та об'єднуючи його з постфіксом ".0", а вхідні дані про сертифікат записуємо в поле "EAC".

```
utxo := UTXO{}
utxo.Key = ctx.GetStub().GetTxID() + ".0"
utxo.Owner = minter
utxo.Amount = amount * 10000 / 10000
utxo.EAC = &eac
```

Кодуємо токен у формат JSON та записуємо його у World State викликаючи `ctx.GetStub().PutState(utxoCompositeKey, tokenJSON)`, по ключу `utxoCompositeKey,` який є складеним та складається з ID користувача (цифрової особи) та ключа токена.

```
// the utxo has a composite key of owner:utxoKey, this enables ClientUTXOs() function to query for an owner's utxos.
utxoCompositeKey, err := ctx.GetStub().CreateCompositeKey("utxo", []string{minter, utxo.Key})

tokenJSON, err := json.Marshal(utxo)
if err ≠ nil {
    return nil, err
}

err = ctx.GetStub().PutState(utxoCompositeKey, tokenJSON)
if err ≠ nil {
    return nil, err
}

log.Printf("utxo minted: %+v", utxo)

return &utxo, nil
```

3.4.1.2 Transfer

Виходи транзакції повинні витрачатися як єдине ціле, оскільки записи в попередніх блоках не можна редагувати (зменшувати). Коли транзакція витрачає UTXO, і користувач не хоче перераховувати всю її суму, надлишок грошей (різниця між розміром UTXO і сумою, яку користувач готовий витратити) надсилається до адреса власника як новий токен.

При оплаті готівкою ви розраховуєте на повернення зміни від свого контрагента. У випадку моделі UTXO одержувач ніколи не контролює решту.

Припустимо, Аліса хоче передати 8 МВт Бобу, і вона контролює один UTXO з amount 10 (сертифікат що підтверджує видобуток 10 МВт). Транзакція, яку вона створює, споживає вихідний UTXO її раніше невитраченої транзакції як вхід. Далі вона визначає, що має статися з її грошима. Вона робить це, створюючи вихідні дані транзакції. Аліса хоче

передати вісім МВт Бобу, тому вона створює два виходи: один платить Бобу, а інший повертає зайві гроші на власну адресу. Для обох виходів умови витрат також визначаються в транзакції, створеній Алісою.

```
// Transfer transfers UTXOs containing tokens from client to recipient(s)
func (s *SmartContract) Transfer(ctx contractapi.TransactionContextInterface, utxoInputKeys []string, utxoOutputs []UTXO) ([]UTXO, error)
```

Функція дістає з бази, валідує та сумує всі токени які подаються на вхід. Так само валідує та сумує всі токени які є виходами транзакції. Далі перевіряється, що сума всіх вхідних токенів дорівнює сумі вихідних і якщо перевірка пройдена - UTXO токени подані на вхід видаляється зі стану їх попереднього власника. Далі аналогічно до функції Mint записується в стан їх отримувача.

3.4.1.3 ClientUTXOs

```
// ClientUTXOs returns all UTXOs owned by the calling client
func (s *SmartContract) ClientUTXOs(ctx contractapi.TransactionContextInterface) ([]*UTXO, error) {
```

Для отримання всіх токенів користувача робиться виклик функції `ctx.GetStub().GetStateByPartialCompositeKey("utxo",[]string{clientID})` яка повертає по складеному ключу ітератор, який можна використовувати для перебору всіх складених ключів, префікс яких відповідає заданому частковому складеному ключу. Перебираючи всі UTXO ключі формуємо самі токени, додаємо їх в масив та повертаємо користувачу.

3.4.2 Реалізація Backend з використанням Nest.js

Основні складові серверної частини застосунку можна розділити на:
модулі, сервіси або провайдери та контролери.

Модуль — це клас, анотований декоратором @Module(). Декоратор @Module() надає метадані, які Nest використовує для організації структури програми.

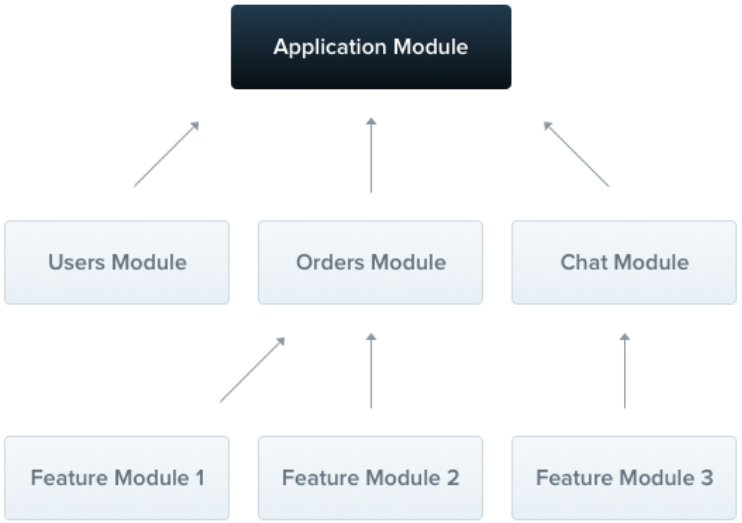


Рис 3.5 - Модуль в Nest.js

Провайдери є ключовим поняттям у Nest. Багато базових класів Nest можуть розглядатися як постачальники – сервіси, репозиторії, фабрики, помічники тощо. Основна ідея провайдера полягає в тому, що його можна **ввести як залежність**; це означає, що об'єкти можуть створювати різні відносини один з одним, а функцію "підключення"

екземплярів об'єктів можна значною мірою делегувати системі середовища виконання Nest.

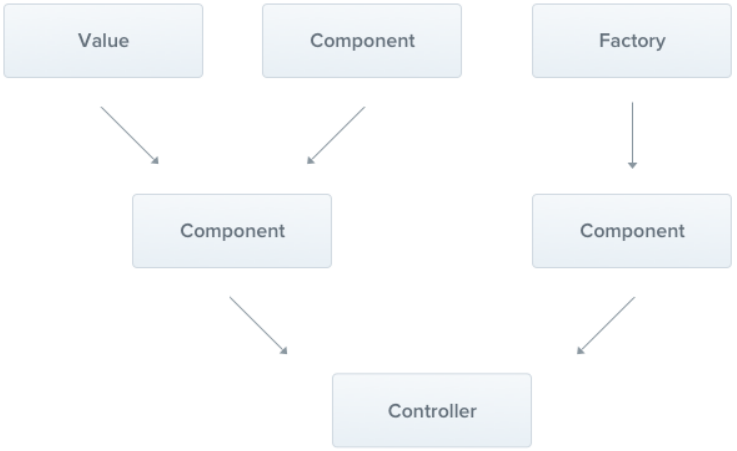


Рис 3.6 - Провайдер в Nest.js

Контролери відповідають за обробку вхідних запитів і повернення відповідей клієнту.

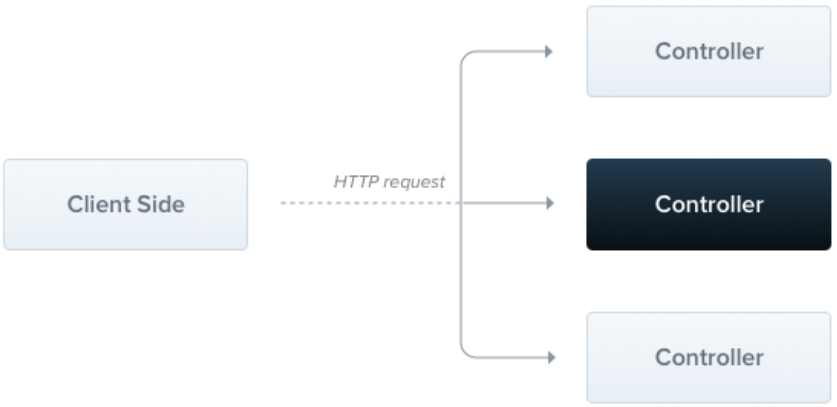


Рис 3.7 - Контроллер в Nest.js

Nest побудовано на основі дуже потужного шаблону проектування, широко відомого як **ін’єкція залежностей**. Ін’єкція залежності, або DI, —

це шаблон проектування, в якому клас запитує залежності із зовнішніх джерел, а не створює їх. Залежності — це сервіси або об’єкти, які потрібні класу для виконання своєї функції.

Впровадження або ін’єкція служб робить їх видимими для компонента. Щоб ввести залежність у компонент необхідно у конструктор компонента передати аргументом тип залежності. Ін’єкція залежностей — це техніка інверсії керування (IoC), при якій ви делегуєте створення екземплярів залежностей контейнеру IoC (у нашому випадку, системі середовища виконання NestJS), замість того, щоб робити це в обов’язковому порядку у власному коді. Для того щоб середовище виконання NestJS зрозуміло, що компонент є провайдером і необхідно буде робити його ін’єкцію, цей компонент необхідно позначити декоратором `@Injectable()`.

Основні модулі застосунку це User, Organisation, Station, Token, EAC Market, Measurements та Wallet.

3.4.2.1 Робота з блокчейн мережею Fabric

Для взаємодії з фабриком використовується модуль `FabricWalletService`, який агрегує два допоміжних сервіси AppService та CaCervice. AppService допомагає побудувати common connection profile для під’єднання у майбутньому до Peer організації 1 за допомогою `Gateway`. CaService інстанціює об’єкт CAClient для взаємодії з центром сертифікації мережі Fabric.

```
@Injectable()
export class FabricWalletService {
    private ccp: any;
    private caClient: any;
    private wallet: undefined;
    private gateway: Gateway;
    private databaseURL = "http://" + process.env.COUCHDB_USER
        + ":" + process.env.COUCHDB_PASSWD
        + "@" + process.env.COUCHDB_HOST
        + ":" + process.env.COUCHDB_PORT;

    constructor(
        private appService: AppService,
        private caService: CaService
    ) {
        this.ccp = appService.buildCCPOrg1();
        this.caClient = caService.buildCAClient(FabricCAServices, this.ccp, process.env.CA_HOST_NAME);
        this.wallet = undefined;
        this.gateway = new Gateway();
    }
```

Для менеджменту X.509 цифрових осіб використовується Singleton об’єкт Wallet. Гаманець дозволяє зберігати сертифікати та закриті ключі користувачі та отримувати їх по унікальному ідентифікатору користувача. Будуємо гаманець методом `createWallet()`, який в свою чергу звертається до агрегованого сервісу AppService та викликає утилітарну функцію побудови гаманця в CouchDB `buildWallet()`.

```
async createWallet() {
    this.wallet = await this.appService.buildWallet(Wallets, this.databaseURL);
    try {
        await this.caService.enrollAdmin(this.caClient, this.wallet, process.env.MSP_ORG);
        await this.caService.registerAndEnrollUser(
            this.caClient,
            this.wallet,
            process.env.MSP_ORG,
            process.env.GARBAGE,
            process.env.AFFILICATION);
    } catch {
        return;
    }
}

async getWallet() {
    if (this.wallet == undefined) {
        await this.createWallet();
    }
    return this.wallet;
}
```

Рис

```
async buildWallet(Wallets, walletPath : string = 'http://admin:adminpw@127.0.0.1:5984') {
    // Create a new  wallet : Note that wallet is for managing identities.
    let wallet;
    if (walletPath) {
        wallet = await Wallets.newCouchDBWallet( config: {
            url: walletPath
        });
    } else {
        wallet = await Wallets.newInMemoryWallet();
        console.log('Built an in memory wallet');
    }

    return wallet;
};
```

Рис

Важливо зазначити, що при побудові гаманця відбувається логін (enrollment) користувача Admin. Це користувач який має право реєструвати нових користувачів за допомогою центру сертифікації.

Далі клас описує селектори для отримання властивостей класу іншими модулями:

```
async getWallet() {
    if (this.wallet === undefined) {
        await this.createWallet();
    }
    return this.wallet;
}

getGateway() {
    return this.gateway;
}

getCCP() {
    return this.ccp;
}

getCaClient() {
    return this.caClient;
}
```

3.4.2.3 Аутентифікація

Аутентифікація є важливою частиною більшості систем. Існує багато різних підходів і стратегій для обробки аутентифікації. Passport — найпопулярніша бібліотека автентифікації в Node.js, добре відома спільноті і успішно використовується в багатьох виробничих програмах. Цю бібліотеку легко інтегрувати з програмою Nest за допомогою модуля `@nestjs/passport`.

На високому рівні Passport виконує ряд кроків для того, щоб:

- Аутентифікувати користувача, перевіривши його "облікові дані" (наприклад, ім'я користувача/пароль, веб-токен JSON (JWT))
- Керувати станом автентифікації (випускаючи портативний маркер, наприклад JWT, або створюючи експрес-сеанс)
- Додавати інформацію про автентифікованого користувача до об’єкта Request для подальшого використання в обробниках маршрутів (id, role)

Passport має багату екосистему **стратегій**, які реалізують різні механізми аутентифікації. Корисно думати про Passport як про міні-фреймворк сам по собі. Елегантність фреймворка полягає в тому, що він абстрагує процес аутентифікації на кілька основних кроків, які ви налаштовуєте відповідно до стратегії, яку ви реалізуєте. Це як фреймворк, оскільки ви налаштовуєте його, надаючи параметри налаштування (як звичайні об’єкти JSON) і користувацький код у вигляді функцій зворотного виклику (callbacks), які Passport викликає у відповідний момент.

AuthService має завдання отримати користувача та перевірити пароль. Для цього створюємо метод validateUser().

```
async validateUser(email: string, pass: string): Promise<any> {
    const user = await this.userService.getUserByEmail(email);
    const isMatch = bcrypt.compareSync(pass, user.password);
    if (user && isMatch) {
        const { password, ...result } = user;
        return result;
    }
    return null;
}
```

В застосунку використовується модуль `bcrypt` для хешування паролів з сіллю. В базі даних паролі користувачів зберігаються в захешованому вигляді.

Далі необхідно реалізувати локальну стратегію аутентифікації.

Реалізуємо метод validate(). Для кожної стратегії Passport викличе функцію verify (реалізована за допомогою методу validate() у @nestjs/passport), використовуючи відповідний набір параметрів для конкретної стратегії. Для локальної стратегії Passport очікує метод validate() з такими аргументами: `validate(username: string, password:string): any.` Метод `validate()` для будь-якої стратегії Passport буде діяти за подібною схемою, відрізняючись лише в деталях представлення облікових даних. Якщо користувача знайдено, а облікові дані дійсні, функція повертає об’єкт користувача, щоб Passport міг виконати свої завдання (наприклад, створити властивість користувача для об’єкта `Request`), і конвеєр обробки запитів може продовжити виконання. Якщо його не знайдено, ми викидаємо помилку і дозволяємо нашому обробнику винятків обробити його [11].

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
    constructor(private authService: AuthService) {
        super({ usernameField: 'email' });
    }

    async validate(username: string, password: string): Promise<any> {
        const user = await this.authService.validateUser(username, password);
        if (!user) {
            throw new UnauthorizedException();
        }
        return user;
    }
}
```

Як правило, єдина істотна відмінність у методі `validate()` для кожної стратегії полягає в тому, як ви визначаєте, чи існує користувач і чи є дійсним. Наприклад, у стратегії JWT, залежно від вимог, ми можемо оцінити, чи відповідає userId, що міститься в декодованому токені, запису в нашій базі даних користувачів, або чи відповідає списку відкликаних токенів. Таким чином, цей шаблон підкласів та реалізації специфічної перевірки стратегії є послідовним, елегантним і розширюваним.

Застосунок може перебувати всього у двух станах:

- Коли користувач аутентифікований
- Та коли ні

Для того, щоб визначити, чи буде запит оброблений обробником маршруту використаємо Guards. Guard — це клас, анотований декоратором @Injectable(), який реалізує інтерфейс CanActivate.

Рис 3.8 - Guards в Nest.js [11]

Створюємо клас який успадковується від бібліотечного класу AuthGuard та вказуємо назву нашої стратегії “local”.

```
@Injectable()
export class LocalAuthGuard extends AuthGuard( type: 'local') {}
```

Використовуємо декоратор @UseGuards() для того щоб застосувати нашу стратегію аутентифікації при зверненні до кінцевої точки `auth/login`.

```
@UseGuards(LocalAuthGuard)
@Post( path: 'login')
async login(@Request() req) {
    return this.authService.login(req.user);
}
```

Це означає що:

- Обробник маршруту буде викликаний, лише якщо користувач був перевірений
- Параметр req міститиме властивість користувача (заповнюється Passport під час процесу локальної аутентифікації)

Далі створюємо метод login який сформує корисне навантаження JWT та підпише його за допомогою секретного ключа.

```
async login(user: any) {
    const payload = { username: user.fullName, sub: user.id, role: user.role };
    return {
        access_token: this.jwtService.sign(payload),
    };
}
```

Секретний ключ вказуємо в модулі Auth.

```
@Module( metadata: {
    imports: [
        ConfigModule.forRoot(),
        UserModule, PassportModule,
        JwtModule.register( options: {
        secret: jwtConstants.secret,
        signOptions: { expiresIn: '24h' },
    })],
    providers: [AuthService, LocalStrategy, JwtStrategy],
    controllers: [AuthController],
    exports: [AuthService]
})
export class AuthModule {
}
```

Тепер кінцева точка `auth/login` повертає при зверненні до неї JWT користувачу.

Захистити кінцеві точки, вимагаючи, щоб у запиті був присутній дійсний JWT можно реалізувавши JWT стратегію та застосувавши декоратор `@UseGuards(JwtAuthGuard)` . `JwtAuthGuard` реалізовано аналогічно до `LocalAuthGuard`.

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
    constructor() {
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            ignoreExpiration: false,
            secretOrKey: jwtConstants.secret,
        });
    }

    async validate(payload: any) {
        //  database lookup if needed
        // looking up the userId in a list of revoked tokens
        return { id: payload.sub, username: payload.username, role: payload.role };
    }
}
```

Для JWT Passport спочатку перевіряє підпис JWT і декодує JSON. Потім він викликає наш метод validate(), передаючи декодований JSON як єдиний параметр. Виходячи з того, як працює підпис JWT, ми гарантуємо, що отримаємо дійсний токен, який ми раніше підписали та видали дійсному користувачеві.

В результаті всього цього наша відповідь на зворотний виклик validate() тривіальна: ми просто повертаємо об’єкт, що містить властивості userId, username та role.

Реєстрація користувача в системі достатньо тривіальна: створюємо в AuthController кінцеву точку POST /auth/register, який приймає CreateUserDto.

```
@Post( path: 'register')
register(@Body() userDto: CreateUserDto) {
  return this.authService.register(userDto);
}
```

Об’єкт передачі даних (DTO) використовується для відокремлення рівня сервісів від рівня бази даних. Це об’єкт, який визначає які дані будуть передані по мережі.

```
export class CreateUserDto {
    @IsEmail()
    @IsNotEmpty( validationOptions: {message: 'Email is required'})
    email: string;

    @IsString()
    fullName: string;

    @IsString()
    @IsNotEmpty( validationOptions: {message: 'Password is required'})
    password: string;

    @IsNotEmpty( validationOptions: { message: 'User role is required' })
    role: UserRole;
}
```

Сам метод у сервісі матиме наступний вигляд:

```
async register(createUserDto: CreateUserDto) {
    try {
        const user = await this.userService.create(createUserDto);
        const registerResult = await this.ca.registerAndEnrollUser(
            this.fsw.getCaClient(),
            await this.fsw.getWallet(),
            process.env.MSP_ORG, user.id.toString(), process.env.AFFILIATION);
        return {
            status: 200,
            message: { user, registerResult },
        };
    } catch (e) {
        return {
            status: e.status,
            message: e.message,
        };
    }
}
```

AuthService в свою чергу агрегує CaService та UserService. UserService, який агрегує репозиторій для роботи з сутністю User у базі даних, використовується для генерування солі та конкатенації її з паролем, створення фіатного гаманця та запису в базу нового користувача. Патерн Репозиторій розглянемо далі.

```
public async create(createUserDto: CreateUserDto): Promise<User> {
    let user;
    const found = await this.userRepository.findOne( options: {
        where: { email: createUserDto.email },
    });
    if (found) throw new UnprocessableEntityException( objectOrError: 'User already exists');
    try {
        const salt = bcrypt.genSaltSync(this.saltRounds);
        const password = bcrypt.hashSync(createUserDto.password, salt);
        user = await this.userRepository.create({ ...createUserDto, password });
        await user.save();
        await this.walletService.create({ userId: user.id, currency: FiatCurrencyEnum.USD, balance: 0 });
        this.logger.verbose( message: 'User added successfully: ', user.id);
    } catch (e) {
        this.logger.error( message: `Failed to add new user: `, e.stack);
        throw new InternalServerErrorException();
    }
    return user;
}
```

Сутність користувач:

```
@Entity()
export class User extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;

    @Column( options: { unique: true })
    email: string;

    @Column()
    fullName: string;

    @Column()
    password: string;

    @Column()
    role: UserRole;
```

```
    @OneToMany( typeFunctionOrTarget: (type) => Organisation, inverseSide: (org : Organisation ) => org.owner, options: {
        eager: false,
    })
    organisations: Promise<Organisation[]>;

    @OneToMany( typeFunctionOrTarget: (type) => EAC, inverseSide: (eac : EAC ) => eac.owner, options: {
        eager: false,
    })
    eacs: Promise<EAC[]>;

    @OneToOne( typeFunctionOrTarget: (type) => Wallet, inverseSide: (wallet : Wallet ) => wallet.owner, options: {
        eager: false,
    })
    wallet: Promise<Wallet>;
}
```

За допомогою CaService реєструємо нову цифрову особу в мережі Fabric викликаючи `registerAndEnrollUser()` та передаючи аргументами Singleton гаманець в який буде додану нову цифрову особу та CA клієнт для взаємодії з центром сертифікації, реалізацію цього методу наведено в Додатку А.

3.4.2.4 Авторизація

Авторизація є чудовим випадком для використання вищеописаних Guards, оскільки конкретні маршрути (кінцеві точки) повинні бути доступні лише тоді, коли клієнт (зазвичай певний автентифікований користувач) має достатні права. AuthGuard, який ми зараз створимо, передбачає автентифікованого користувача (і, отже, токен додається до заголовків запиту). AuthGuard використає інформацію з токена, щоб визначити, чи потрібно виконати запит чи ні.

Створюємо клас RolesGuard який реалізує інтерфейс CanActivate.

```
@Injectable()
export class RolesGuard implements CanActivate {
    constructor(private reflector: Reflector) {}

    canActivate(context: ExecutionContext): boolean {
        const roles = this.reflector.get<string[]>( metadataKey: 'roles', context.getHandler());
        if (!roles) {
            return true;
        }
        const request = context.switchToHttp().getRequest();
        const user = request.user;
        return roles.includes(user.role);
    }
}
```

Кожен Guard повинен реалізувати функцію canActivate(). Ця функція повинна повертати логічне значення, яке вказує, чи дозволено обробку поточного запиту чи ні.

Nest надає можливість приєднувати власні метадані до обробників маршрутів через декоратор @SetMetadata(). Ці метадані забезпечують наші відсутні дані про роль, які потрібні розумному “охоронцю” для

прийняття рішень. Для того щоб чисто, читабельно, істрого типізовано визначити в контроллері які ролі матимуть доступ до маршруту, створюємо окремий декоратор:

```
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles: string[]) => SetMetadata( metadataKey: 'roles', roles);
```

Залишається застосувати наш RolesGuard в необхідному контролері:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Controller( prefix: 'organisation')
```

Та описати ролі які мають доступ до певних маршрутів :

```
@Get()
@Roles(UserRole.OrganisationOwner)
getAllOrganisations(@Req() req): Promise<Organisation[]> {
    return this.organisationService.getAllOrganisations(req.user.id);
}
```

3.4.2.5 Робота з базою даних PostgreSQL

Nest не залежить від бази даних, що дозволяє легко інтегруватися з будь-якою базою даних SQL або NoSQL. У вас є кілька доступних варіантів, залежно від ваших уподобань. На найзагальнішому рівні

підключення Nest до бази даних — це просто завантаження відповідного драйвера Node.js для бази даних.

Щоб почати його використовувати, ми спочатку встановлюємо необхідні залежності.

```
$ npm install --save @nestjs/typeorm typeorm@0.2 pg
```

Після завершення процесу встановлення ми можемо імпортувати TypeOrmModule в кореневий AppModule.

```
@Module( metadata: {
    imports: [
        ConfigModule.forRoot( options: { isGlobal: true }),
        TypeOrmModule.forRoot(typeOrmConfig),
        OrganisationModule, StationModule, EacMarketModule, MeasurementsModule,
        WalletModule, UserModule, TokenModule, AuthModule
    ],
    controllers: [AppController],
    providers: [AppService],
})
export class AppModule {
}
```

Для встановлення з’єднання з базою даних необхідно створити конфігураційний файл, в якому вказати тип бази даних, адресу та порт, ім’я користувача та пароль, а також назву самої бази та сутності з якими необхідно працювати.

```
export const typeOrmConfig: TypeOrmModuleOptions = {
    type: 'postgres',
    host: 'localhost',
    port: 5432,
    username: 'postgres',
    password: 'postgres',
    database: 'cw2022',
    entities: [__dirname + '/../**/*.entity.{js,ts}'],
    synchronize: true,
};
```

TypeORM підтримує шаблон проектування Репозиторій, тому кожна сутність має свій власний репозиторій. Ці репозиторії можна отримати з об’єкту Connection. Цей шаблон надає можливість абстрагуватися від конкретної бази даних та розділити обов’язки між компонентами, які працюють з даними, та з рештою логіки застосунку. Це зменшує зв’язність компонентів застосунку та підвищує згуртованість коду.

3.4.2.6 Реалізація модулів Organisation, Station, Measurements

Сам модуль має імпортувати репозиторій та експортувати сервіс, який нам знадобиться в Station:

```
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [OrganisationRepository]), UserModule],
  controllers: [OrganisationController],
  providers: [OrganisationService],
  exports: [OrganisationService]
})
export class OrganisationModule {}
```

Клас успадкований від Repository описує всі основновні дії із базою даних: insert, find, save. Створений клас репозиторію має наступний вигляд:

```
import { EntityRepository, Repository } from 'typeorm';
import { Organisation } from './entities/organisation.entity';

@EntityRepository(Organisation)
export class OrganisationRepository extends Repository<Organisation> {}
```

В класі `Organisation` який успадковується від класу `BaseEntity` описуємо відображення (mapping) таблиці з БД.

```
@Entity()
export class Organisation extends BaseEntity {
    @PrimaryColumn()
    registryNumber: number;

    @Column()
    name: string;

    @Column()
    ownerId: number;

    @Column()
    businessType: string;

    @Column()
    organisationAddress: string;

    @Column()
    organisationEmail: string;

    @OneToMany( typeFunctionOrTarget: (type) => Station, inverseSide: (station : Station ) => station.organisation, options: {
        eager: false,
    })
    @JoinTable()
    stations: Promise<Station[]>;
```

Декоратор @Entity() використовується для позначення класів, які будуть сутністю (таблиця або документ в залежності від типу бази даних). Схема бази даних буде створена для всіх класів, декорованих @Entity(), далі репозиторій можна отримати та використовувати для цієї сутності.

Окрім того в цьому класі описуємо типи зв’язків з іншими сутностями. Наприклад, декоратор @OneToMany(()=>Station) описує зв’язок “один до багатьох” між сутностями Організація та Станція.

Більш високорівнева робота з колекціями інкапсулюється у сервісі. Сервіси агрегують репозиторій для роботи з відповідною сутністю та інші сервіси за необхідністю. Середовище Nest робить за нас ін’єкцію вказаної залежності.

```
constructor(
    @InjectRepository(OrganisationRepository)
    private organisationRepository: OrganisationRepository,
) {
}
```

Щоб надати користувачу можливість створити організацію, пишемо наступний метод:

```
public async createOrganisation(
    organisationInput: CreateOrganisationDto,
    userId: number,
): Promise<Organisation> {
    let organisation;
    let found = await this.organisationRepository.findOne( options: {
        where: { registryNumber: organisationInput.registryNumber },
    });
    if (found) throw new UnprocessableEntityException( objectOrError: 'Organisation already exists');
    try {
        organisation = this.organisationRepository.create(organisationInput);
        organisation.ownerId = userId;
        await organisation.save();
    } catch (error) {
        this.logger.error( message: `Failed to create organisation ${organisationInput.registryNumber}`, error);
        throw new InternalServerErrorException( objectOrError: 'Organisation creation failed');
    }
    return organisation;
}
```

Для отримання всіх організацій власником яких є користувач пишемо наступний метод:

```
public async getAllOrganisations(userId: number): Promise<Organisation[]> {
    let found;
    const user = await this.userService.getUserById(userId);
    try {
        found = await user.organisations;
    } catch (error) {
        this.logger.error( message: `Failed to get all stations: `, error.stack);
        throw new InternalServerErrorException();
    }
    if (!found || found.length === 0) {
        throw new NotFoundException( objectOrError: `Organisations not found`);
    }
    return found;
}
```

В окремому класі створюємо Controller для обробки клієнтських запитів. В OrganisationController робимо ін'єкцію нашого сервісу, в якому описана основна логіка.

```
constructor(private readonly organisationService: OrganisationService) {}
```

Контролер лише делегує виконання сервісу. POST /organisation приймає аргументом DTO та id користувача, який за нас в HTTP запит записав JwtAuthGuard.

```
@Post()
@Roles(UserRole.OrganisationOwner)
createOrganisation(
    @Body(ValidationPipe) createOrganisationDto: CreateOrganisationDto,
    @Req() req,
): Promise<Organisation> {
    this.logger.verbose(
        message: `Creating new Organisation. Data : ${JSON.stringify(createOrganisationDto)}`,
    );
    return this.organisationService.createOrganisation(createOrganisationDto, req.user.id);
}
```

Фрагмент CreateOrganisationDto:

```
export class CreateOrganisationDto {
    @IsNotEmpty( validationOptions: { message: 'Name should not be empty' })
    @IsString()
    @MaxLength( max: 50,  validationOptions: {
        message: 'Name must be shorter than or equal to 50 characters',
    })
    name: string;

    @IsNotEmpty( validationOptions: { message: 'Trade register number should not be empty' })
    @IsNumber()
    registryNumber: number;

    @IsNotEmpty( validationOptions: { message: 'Business Type should not be empty' })
    @IsString()
    @MaxLength( max: 50,  validationOptions: {
        message: 'Business Type must be shorter than or equal to 50 characters',
    })
    businessType: string;
```

DTO зручні тим, що ми декларативно вказуємо які поля мають міститися в запиті та їх тип. Всі перевірки за нас робить середовище Nest, а саме ValidationPipe з модулю class-validator.

Реалізація модулів Station та Measurements загалом аналогічна. Потрібно зауважити лише те, що при створенні електростанції екземпляр сутності Entity створюється також в контекстному брокері Fiware Orion. За допомогою бібліотеки axios надсилаємо POST запит на створення контексту. ID станції в контекстному брокері є складеним та містить реєстраційний номер компанії та назву станції.

```
try {
    station.organisation = Promise.resolve(organisation);
    const response = await axios.post(
        url: `${process.env.BROKER_HOST}:${process.env.BROKER_PORT}/v2/entities?options=keyValues`,
        data: {
            type: 'Station',
            id: `${station.organisationRegistryNumber}.${station.name.replace( searchValue: / /g,  replaceValue: '_')}`,
            startDate: '',
            endDate: '',
            generatedEnergy: '0',
        },
        config: { headers: { 'Content-Type': 'application/json' } },
    );
    this.logger.verbose( message: 'Create Station in Context Broker status: ', response.status);
    await station.save();
} catch (error) {
```

3.4.2.7 Реалізація модуля Token

Модуль Token присвячено роботі з UTXO токеном, а саме його створенню, трансферу та отриманню токенів користувача.

```
@Module( metadata: {
  controllers: [TokenController],
  imports: [FabricModule, TokenUtilsModule, StationModule, MeasurementsModule],
  providers: [TokenService],
  exports: [TokenService]
})
export class TokenModule {}
```

Об’єкт класу CreateTokenDto містить всього одне обов’язкове поле - stationId, оскільки всі дані які він має містити отримуються з бази даних.

```
export class CreateTokenDto {
    @IsString()
    @IsNotEmpty( validationOptions: { message: 'Station ID is required' })
    stationId: string;
}
```

Спочатку маємо отримати об’єкт Station з бази даних:

```
const { stationId } = createTokenDto;
const [organisation, name] = stationId.split( separator: '.');
const station = await this.stationService.getStationById(+organisation, name, userId);
const {
    countryId,
    regionId,
    stationEnergyType,
    manufacturerCountryId,
    manufactureDate,
    commissioningDate,
    plantPerformance,
} = station;
```

Після чого отримуємо всі показники по цій станції, фільтруючи ті, які ще не були враховані при попередніх випусках токенів. Крім цього, рахуємо загальну кількість виробленої електроенергії

```
const measurementsArray = (await station.measurements).filter(
    m => m.minted === false && m.generatedEnergy > 0
);
if (measurementsArray.length === 0) {
    throw {
        status: 404,
        message: 'No measurements yet',
    };
}
const generatedEnergy = measurementsArray
    .reduce((acc : number , curr : Measurement ) => acc + curr.generatedEnergy, 0);
```

Далі формуємо сам EAC, отримуємо контракт з мережі блокчейн та надсилаємо транзакцію Mint, передаючи аргументами `totalAmount` та об’єкт EAC^

```
const EAC = {
    prod_start_date: measurementsArray[0].startDate,
    prod_end_date: measurementsArray[measurementsArray.length - 1].endDate,
    generated_energy: generatedEnergy,
    station_uid: stationId,
    station_location: countryId + '.' + regionId,
    station_energy_type: stationEnergyType,
    manufacturer_country_id: manufacturerCountryId,
    manufacture_date: manufactureDate,
    commissioning_date: commissioningDate,
    plant_performance: plantPerformance,
};

const totalAmount = generatedEnergy * 10 / 10000;
const contract = await getContractForUser(this.fws, userId);
await contract.submitTransaction( name: 'Mint', String(totalAmount),
    JSON.stringify(EAC));
```

Після чого залишається лише оновити статуси всіх показників електростанції:

```
for (const m of measurementsArray) {
    // @ts-ignore
    await this.measurementsService.update(m.id, measurement: { ...m, minted: true });
}
```

Окремої уваги заслуговує утилітарна функція getContractForUser(), яка необхідна для створення з'єднання з блокчейн мережею, отримання необхідного каналу та чейнкоду.

```
export async function getContractForUser(fws, userId) {
    await fws.getGateway().connect(fws.getCCP(), {
        wallet: await fws.getWallet(),
        identity: userId.toString(),
        discovery: { enabled: true, asLocalhost: true },
    });
    const network = await fws.getGateway().getNetwork(process.env.CHANNEL_NAME);
    return network.getContract(process.env.CHAINCODE_NAME);
}
```

Для передачі токена необхідно отримати унікальний ключ користувача (цифрової особи) в мережі блокчейн. Для цього робимо виклик утилітарної функції з UserService:

```
async getUserClientId(username) {
    const contract = await getContractForUser(this.fws, username);
    return bufferToString(await contract.evaluateTransaction( name: "ClientID"));
}
```

Після чого необхідно відправити транзакцію в мережу для отримання всіх токенів користувача:

```
async transferToken({ userId, recipientId, tokenId, transferAmount: number = 0 }): Promise<any[]> {
    const recipientKey = await this.userUtils.getUserClientId(recipientId.toString());
    const contract = await getContractForUser(this.fws, userId);
    const userAllTokens = bufferToObject(await contract.evaluateTransaction( name: 'ClientUTXOs'));
    const tokenIndex = userAllTokens.findIndex(token => token.utxo_key == tokenId);
    const { utxo_key: userTokenKey, owner, amount: tokenAmount, EAC } = userAllTokens[tokenIndex];
```

Далі ми формуємо новий стан активів користувача, невитрачені виходи транзакцій токена:

```
let transferResult = [];
const amountLeft = tokenAmount - transferAmount;
if (amountLeft != 0 && transferAmount != 0) {
    if (amountLeft < 0) {
        throw {
            status: 400,
            message: 'Transfer amount exceeds token amount',
        };
    } else {
        transferResult = [
            {
                Key: '',
                Owner: recipientKey,
                Amount: +transferAmount,
                EAC: EAC,
            },
            {
                Key: '',
                Owner: owner,
                Amount: (tokenAmount * 1000 - transferAmount * 1000) / 1000,
                EAC: EAC,
            },
        ];
    }
} else {
```

```
} else {
    transferResult.push({
        Key: '',
        Owner: recipientKey,
        Amount: tokenAmount,
        EAC: EAC,
    });
}
const data = await contract.submitTransaction(
    name: 'Transfer',
    JSON.stringify( value: [userTokenKey]),
    JSON.stringify(transferResult));
return bufferToString(data) === '' ? [] : bufferToObject(data);
```

В залежності від того який amount користувач хоче передати іншому користувачу, в результаті буде 1 або 2 UTXO токени. Метадані токена, тобто сам сертифікат енергетичних атрибутів, копіюється в новий токен.

Нарешті відправляємо транзакцію для підтвердження у мережу.

Для взаємодії з TokenService, TokenController має бути реалізований наступним чином:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Controller( prefix: 'token')
export class TokenController {
  constructor(private readonly tokenService: TokenService) {}

  @Post()
  @Roles(UserRole.OrganisationOwner)
  create(@Body() createTokenDto: CreateTokenDto, @Req() req) {
    return this.tokenService.create(createTokenDto, req.user.id);
  }

  @Get()
  @Roles(UserRole.OrganisationOwner)
  findAll(@Req() req) {
    return this.tokenService.getClientUTXOs(req.user.id);
  }

  @Put()
  @Roles(UserRole.OrganisationOwner)
  transfer(@Req() req: Request) {
    return this.tokenService.transferByKeyAndAmount(req.body);
  }

  @Delete()
  @Roles(UserRole.OrganisationOwner)
  redeem(@Req() req: Request) {
    return this.tokenService.redeem(req);
  }
}
```

3.4.2.8 Реалізація модуля EAC Market

Сам модуль що описує залежності маркету:

```
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [EacRepository]), WalletModule, TokenModule, UserModule],
  controllers: [EacMarketController],
  providers: [EacMarketService],
  exports: [EacMarketService]
})
export class EacMarketModule {}
```

EacMarketService агрегує власний репозиторій та наступні модулі, що ми реалізували раніше:

```
constructor(
    @InjectRepository(EacRepository)
    private eacRepository: EacRepository,
    private userService: UserService,
    private tokenService: TokenService,
    private walletService: WalletService,
) {
}
```

Для виставлення токену на продаж необхідно перевірити чи не виставлено його вже на ринку та чи є взагалі вказаний токен у користувача.

```
public async sellEac(userId: number, createEacDto: CreateEacDto): Promise<EAC> {
    let eac;
    const { tokenId } = createEacDto;
    let found = await this.eacRepository.findOne( options: {
        where: { userId, tokenId },
    });
    if (found) throw new UnprocessableEntityException( objectOrError: 'EAC is already on market');
    let token = (await this.tokenService.getClientUTXOs(userId)).find(token => token.utxo_key === tokenId);
    if (!token) throw new NotFoundException( objectOrError: 'Token not found');
    try {
        eac = this.eacRepository.create({ ...createEacDto, userId });
        await eac.save();
        return eac;
    } catch (error) {
        this.logger.error( message: `Failed to create new EAC ${tokenId} on market for ${userId}`, error);
        throw new InternalServerErrorException( objectOrError: 'EAC creation failed');
    }
}
```

Далі створюємо об’єкт в базі даних, який зберігає дані про те хто виставив токен на продаж, який саме токен та за якою ціною.

```
@Entity()
export class EAC extends BaseEntity {
    @PrimaryColumn()
    userId: number;

    @PrimaryColumn()
    tokenId: string;

    @Column()
    price: number;

    @ManyToOne( typeFunctionOrTarget: (type) => User, inverseSide: (user : User ) => user.eacs, options: {
        eager: false,
        onDelete: 'CASCADE',
    })
    @JoinColumn( options: { name: 'userId' })
    owner: Promise<User>;
}
```

Для реалізації механізму придбання токена необхідно виконати наступні кроки:

1. Знайти необхідний виставлений на продаж токен в базі маркету
2. Перевірити фіатний баланс покупця
3. Зняти обраховану кількість фіату з балансу покупця
4. Передати токен покупцю
5. Поповнити фіатний баланс продавця
6. Оновити дані про виставлений на продаж сертифікат (видалити якщо куплено повністю, інакше записати новий UTXO ключ токена)

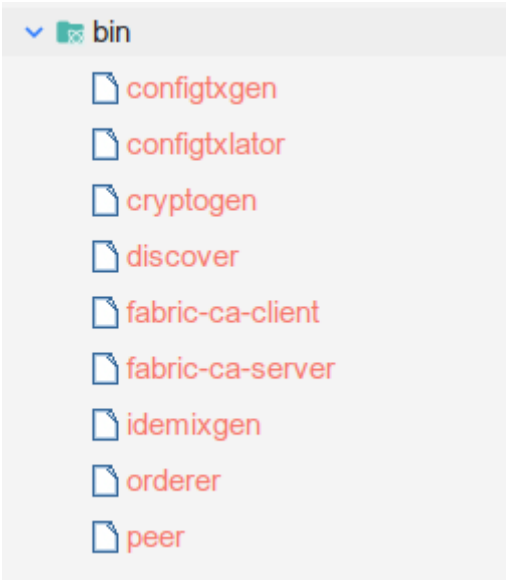
Повний код методу наведено в Додатку Б.

Для взаємодії з сервісом Маркету пишемо класс EacMarketController, основні шляхи якого описано нижче:

```
@UseGuards(JwtAuthGuard)
@Controller( prefix: 'eac-market')
export class EacMarketController {

    constructor(private readonly eacService: EacMarketService) {
    }

    @Post( path: 'sell')
    create(@Body() createEacDto: CreateEacDto, @Req() req) {
        return this.eacService.sellEac(req.user.id, createEacDto);
    }

    @Post( path: 'buy')
    buy(@Req() req) {
        const { userId, tokenId, amount } = req.body;
        return this.eacService.buyEac(req.user.id, userId, tokenId, amount);
    }
```

3.4.3 Розгортання мережі Fabric

Для безпосереднього підняття мережі використаємо скрипт у файлі docker-compose-test-net.yaml. Для конфігурації мережі після її розгортання використовуються згенеровані бінарні файли.



Виконуємо команду яка створює канал с назвою channel1. Криптоматеріали для учасників мережі в цьому каналі будут згенеровані автоматично сертифікаційним центром. База даних для зберігання цифрових осіб, їх сертифікатів та активів - couchdb

```
./network.sh up createChannel -ca -c channel1 -s couchdb -verbose
```

Далі виконуємо команду яка розгорне Chaincode за вказаним шляхом в channel1. Мова програмування - Go.

```
./network.sh deployCC -c channel1 -ccn token_utxo_extended -ccp
../contracts/token-utxo-extended/chaincode-go/ -ccl go
```

Для оновлення версії чейнкоду не потрібно наново розгортати мережу, а необхідно виконати наступну команду, яка є дуже схожою до попередньої, вказуємо лише нове chaincode sequence число та версію контракта.

```
./network.sh deployCC -c channel1 -ccn token_utxo_extended -ccp
../contracts/token-utxo-extended/chaincode-go/ -ccl go -ccs 2
-ccv 2.0
```

Для зупинки мережу виконується команда `./network.sh down.`

Результат виконання скрипта для запуску мережі:

```
dkholodov@dk-blade15:~/Study/cw-2022/test-network$ docker ps -a
CONTAINER ID   IMAGE
                          COMMAND                  CREATED
               NAMES
eaf6c6ff4758   dev-peer0.org1.example.com-token_utxo_extended_1.0-2
13aad24e176ee37829c0dbcf5a58   "chaincode -peer.add…"   3 days ago
               dev-peer0.org1.example.com-token_utxo_extended_1.0-2
e19a440ddc16   dev-peer0.org2.example.com-token_utxo_extended_1.0-2
ff4dc83e4f376e9b240c9cb5a7ce   "chaincode -peer.add…"   3 days ago
               dev-peer0.org2.example.com-token_utxo_extended_1.0-2
e62020109fe4   hyperledger/fabric-tools:2.2.5
                          "/bin/bash"              3 days ago
               cli
6bdb78b356ad   hyperledger/fabric-peer:2.2.5
                          "peer node start"        3 days ago
cp             peer0.org1.example.com
0a866cdf663f   hyperledger/fabric-peer:2.2.5
                          "peer node start"        3 days ago
1->19051/tcp   peer0.org2.example.com
c2d830e2b249   couchdb:3.1.1
                          "tini -- /docker-ent…"   3 days ago
               couchdb1
60ed01c5cc9f   hyperledger/fabric-orderer:2.2.5
                          "orderer"                3 days ago
cp             orderer.example.com
21a6fe5082d7   couchdb:3.1.1
                          "tini -- /docker-ent…"   3 days ago
               couchdb0
18e5371396fd   hyperledger/fabric-ca:1.5.2
                          "sh -c 'fabric-ca-se…"   3 days ago
4->19054/tcp   ca_orderer
0c285d6a5916   hyperledger/fabric-ca:1.5.2
                          "sh -c 'fabric-ca-se…"   3 days ago
4->18054/tcp   ca_org2
0a3acc34deef   hyperledger/fabric-ca:1.5.2
                          "sh -c 'fabric-ca-se…"   3 days ago
cp             ca_org1
5e7737cef4c0   fiware/orion:1.4.1
                          "/usr/bin/contextBro…"   7 days ago
```

Рис 3.9 - блокчейн мережа

3.4.3.1 Опис блокчейн мережі

Кожна нода і користувач, які взаємодіють з мережею Fabric, повинні належати до організації, щоб брати участь у мережі. Тестова мережа включає дві організації, Org1 і Org2. В мережі також розгортається одна служба упорядкування (Ordering Service), яка виконує задачу впорядкування транзакцій в мережі.

Peers є основними компонентами будь-якої мережі Fabric. Піри зберігають леджер блокчейну та перевіряють транзакції, перш ніж вони будуть записані в реєстр. Піри виконують бізнес-логіку для управління активами в леджері, прописану в смарт-контрактах.

Кожен пір у мережі повинен належати до організації. У тестовій мережі кожна організація оперує одним вузлом (Peer), peer0.org1.example.com і peer0.org2.example.com.

Кожна мережа Fabric також містить Ordering Service. У той час як піри перевіряють транзакції та додають блоки транзакцій до блокчейну, вони не визначають порядок транзакцій і не включають їх у нові блоки. У розподіленій мережі однорангові вузли можуть знаходитися далеко один від одного і не мати спільного уявлення про те, коли була створена транзакція. Досягнення консенсусу щодо порядку транзакцій – це дорогий процес, який створить накладні витрати для пірів.

Ordering Service дає змогу пірам зосередитися на перевірці транзакцій та занесенні їх до книги. Після того, як вузли упорядкування отримують схвалені транзакції від клієнтів, вони приходять до консенсусу

щодо порядку транзакцій, а потім додають їх до блоків. Потім блоки розподіляються між одноранговими вузлами (пірами), які в свою чергу додають блоки до блокчейну.

Для досягнення консенсусу про порядок транзакцій у мережі використовується алгоритм Raft. [13]

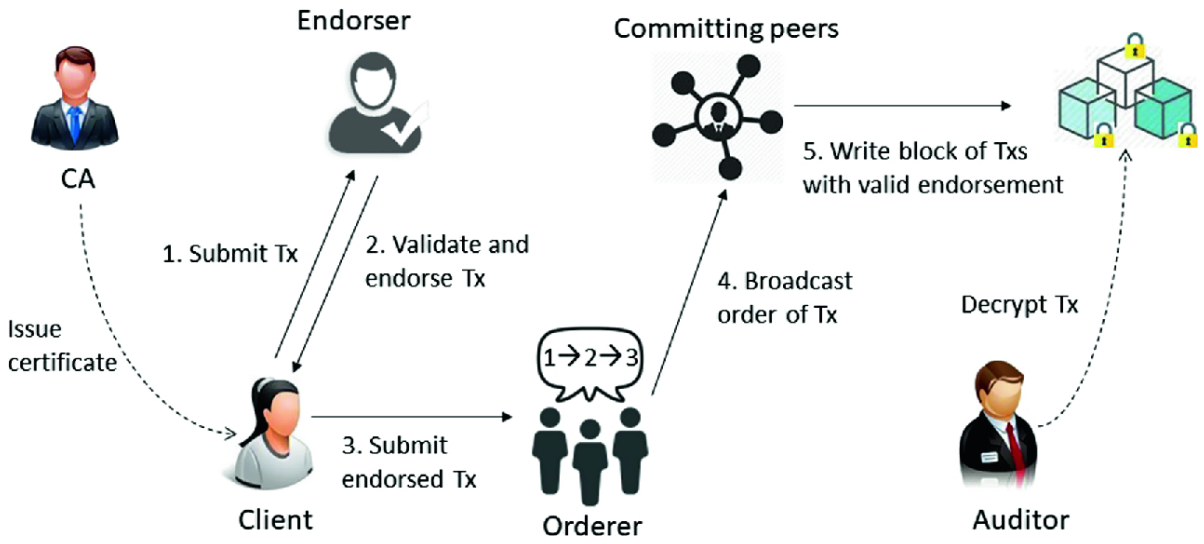


Рис 3.10 - консенсус у Fabric[12]

3.5 Тестування програми і результати її виконання

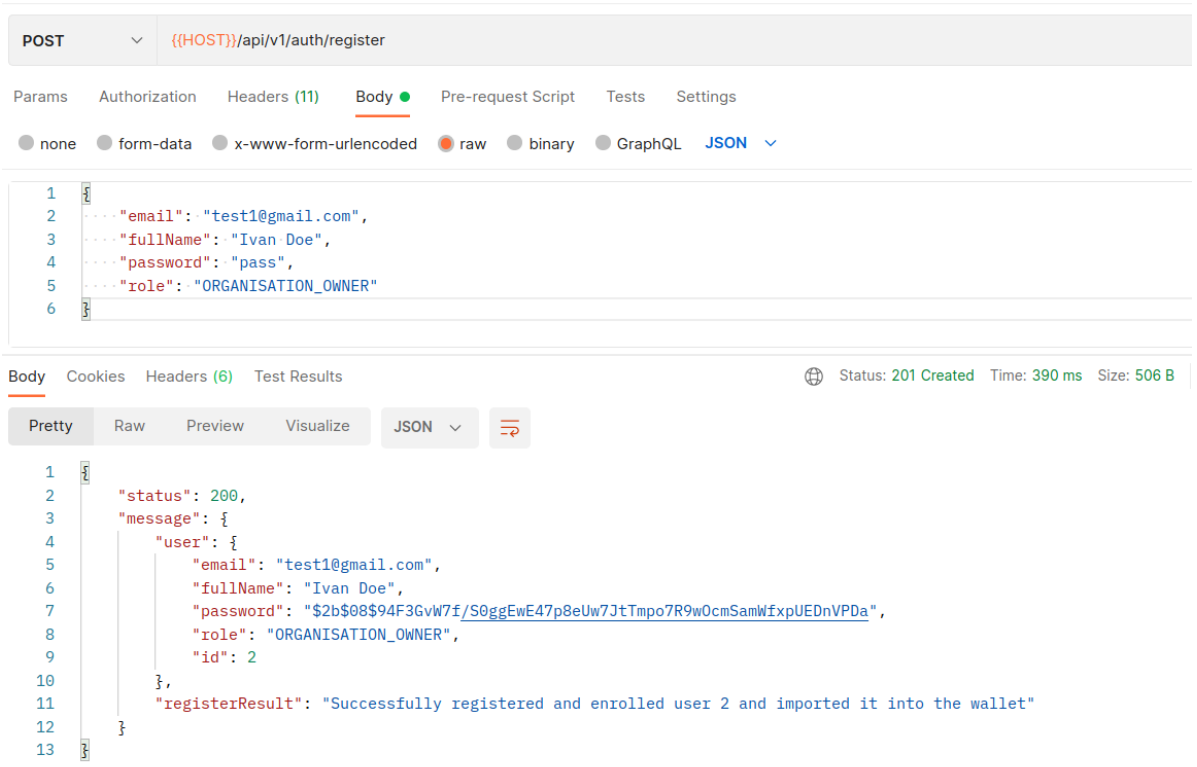


Рис 3.11 - реєстрація власника компанії

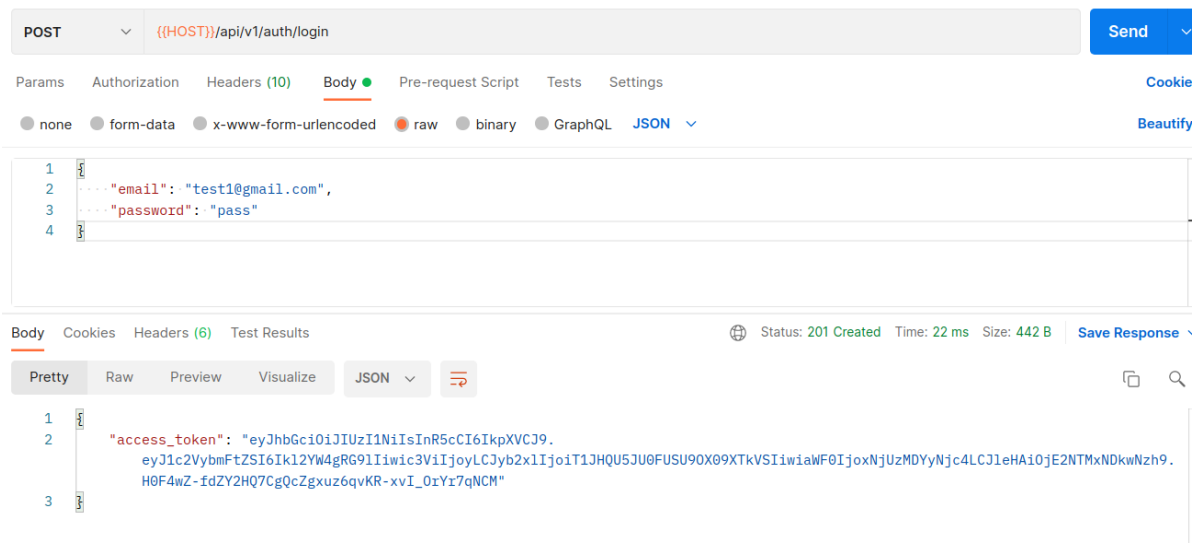


Рис 3.12 - аутентифікація користувача

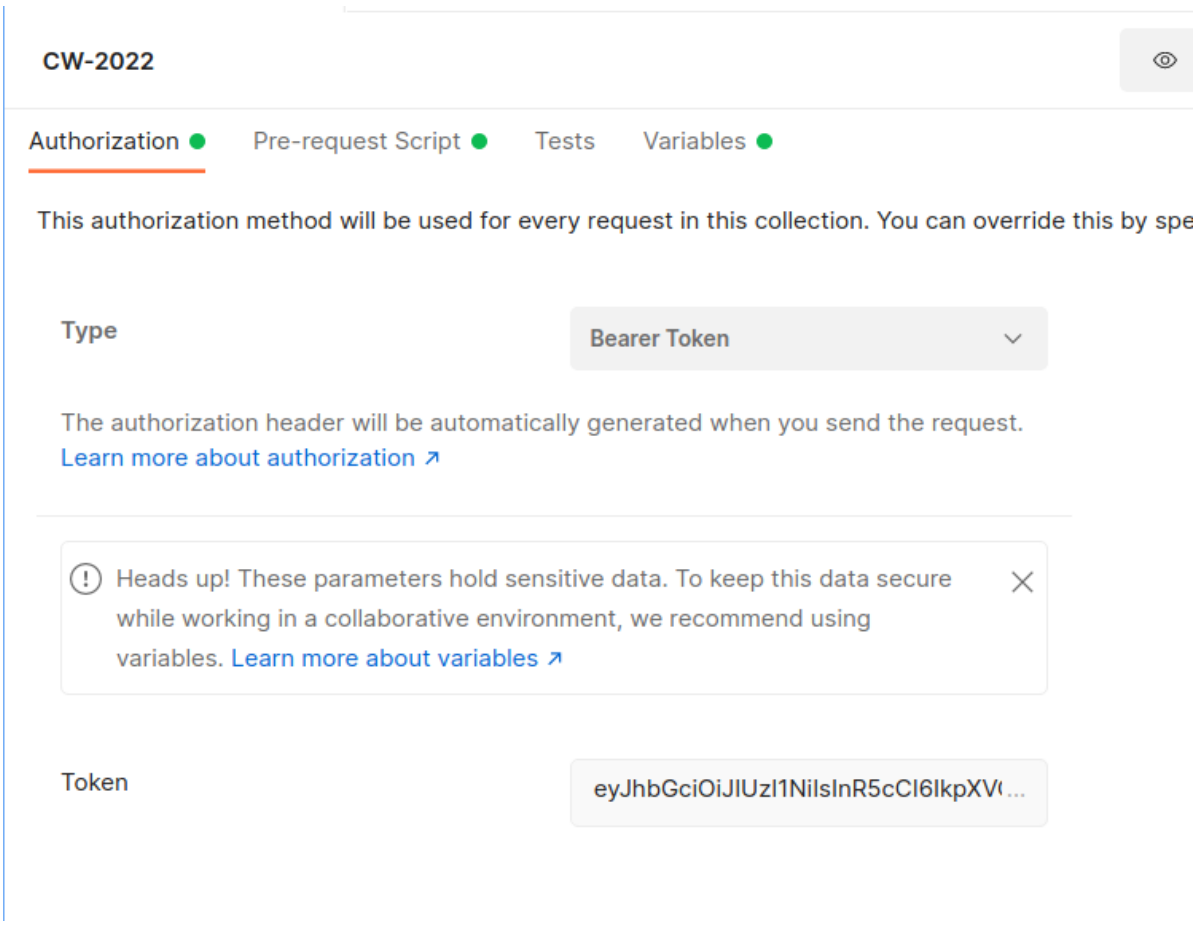


Рис 3.13 - JSON Web Token

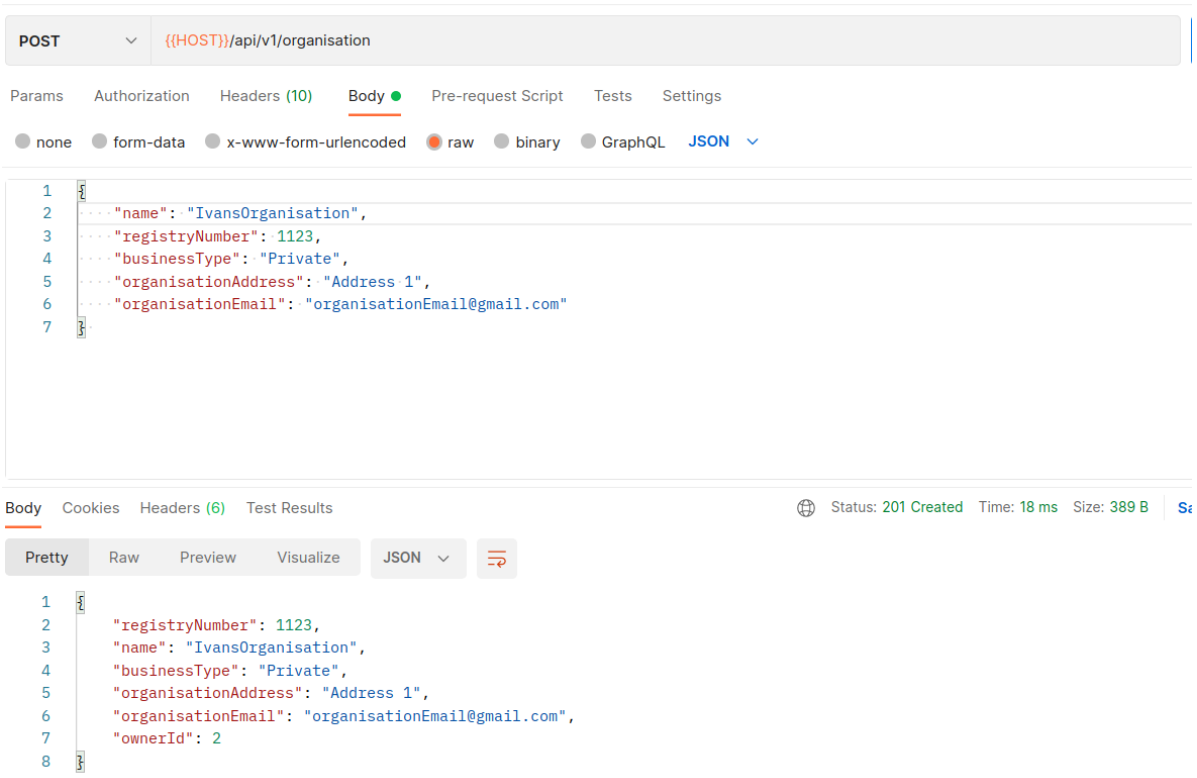


Рис 3.14 - створення організації

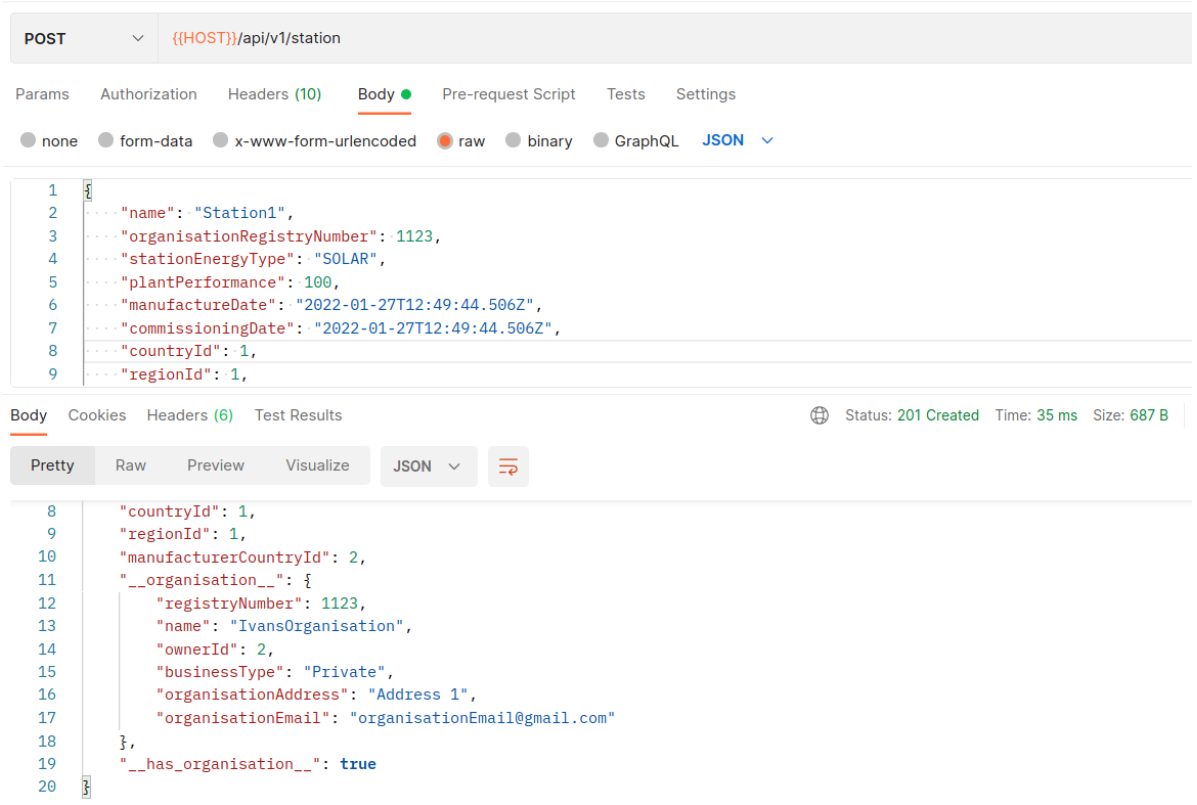


Рис 3.15 - створення електростанції

Бачимо що станцію створено в базу контекстного брокеру.

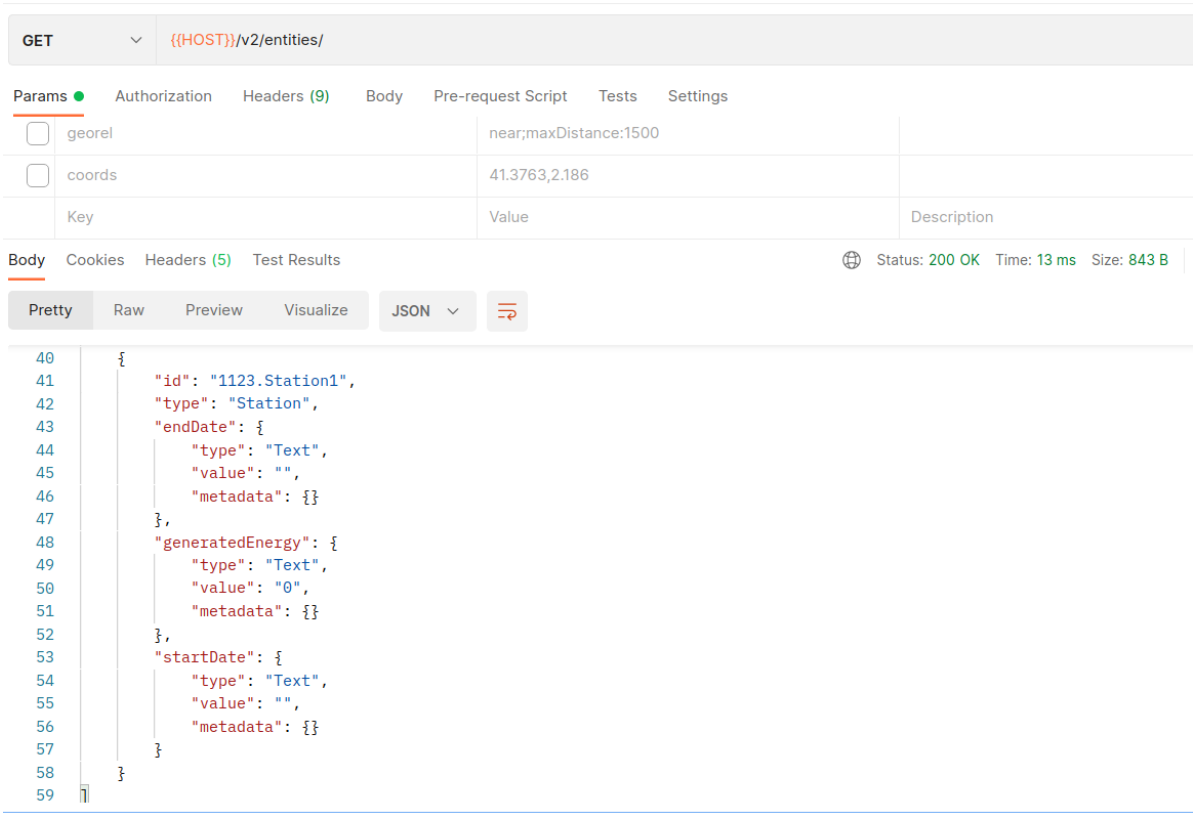


Рис 3.16 - сутності в контекстному брокері

Робимо два запити, що симулюють два виміри виробленої електроенергії з IoT пристроїв.

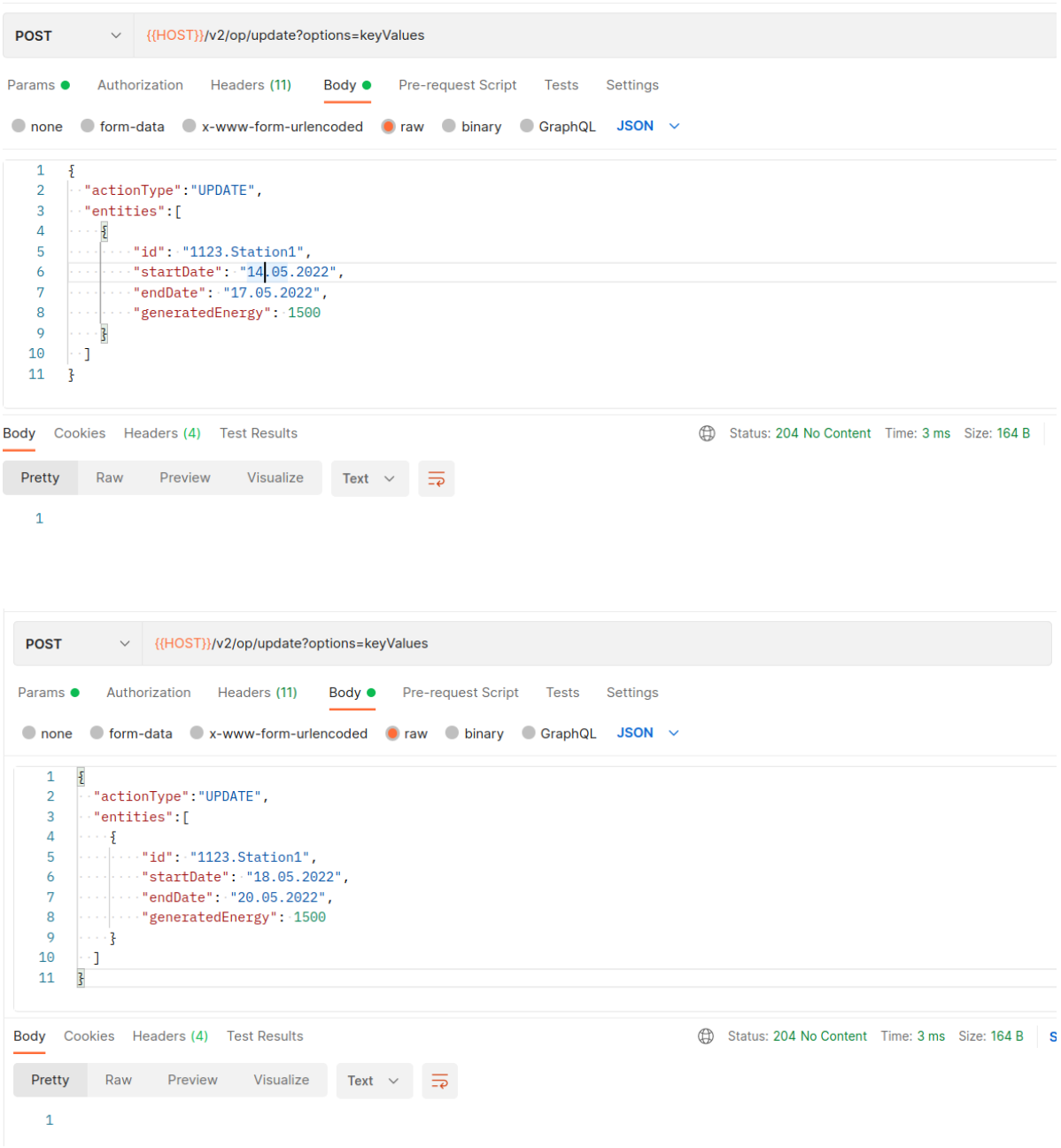


Рис 3.17 - два виміри виробленої е/е

Бачимо що було відправлено сповіщення з контекстного брокеру на бекенд сервіс за допомогою механізму підписки.

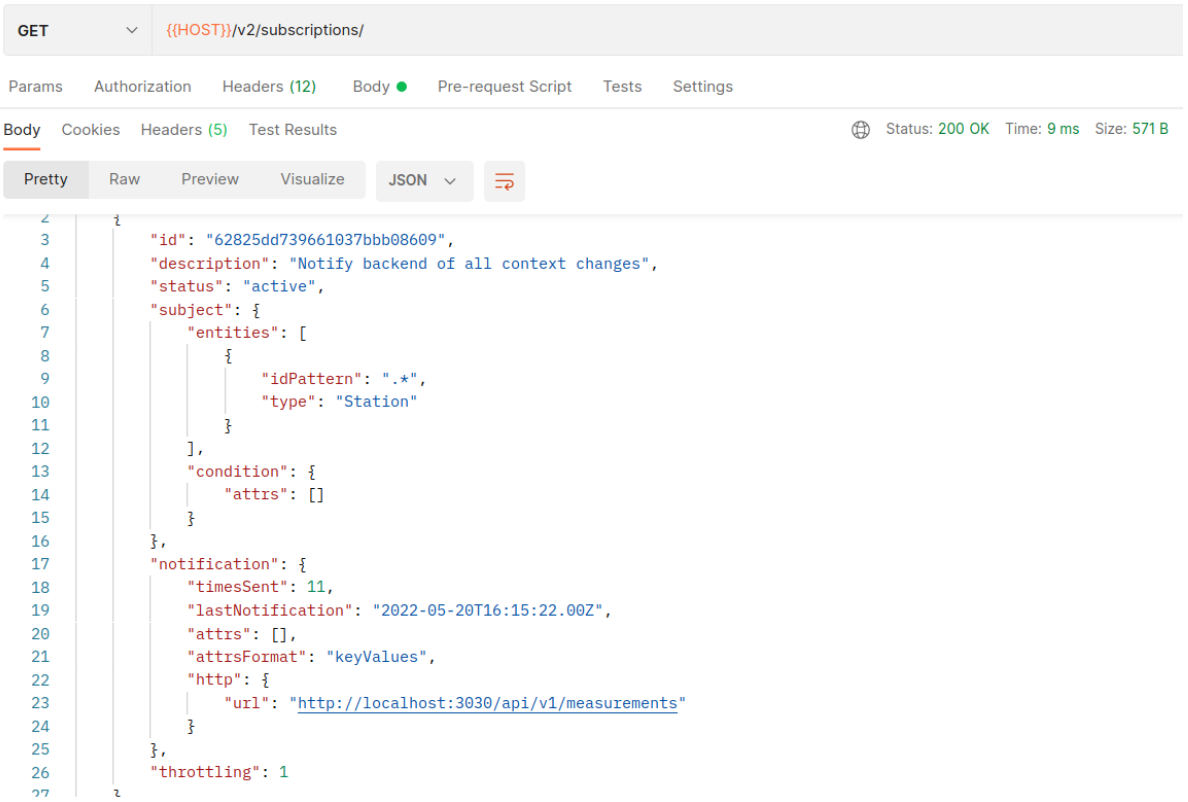


Рис 3.18 - підписка

Контекст станції оновився.

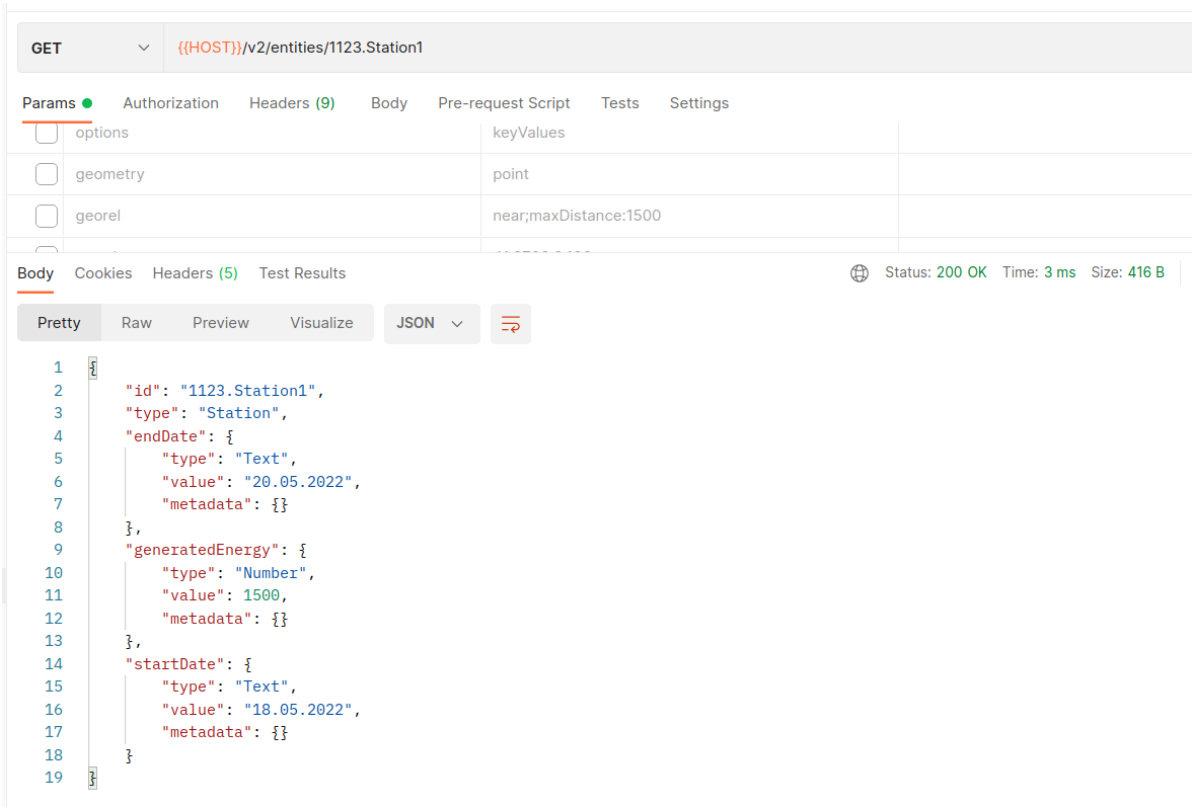


Рис 3.19 - оновлення конексту

Зчитуємо виміри по станції 1123.Station1

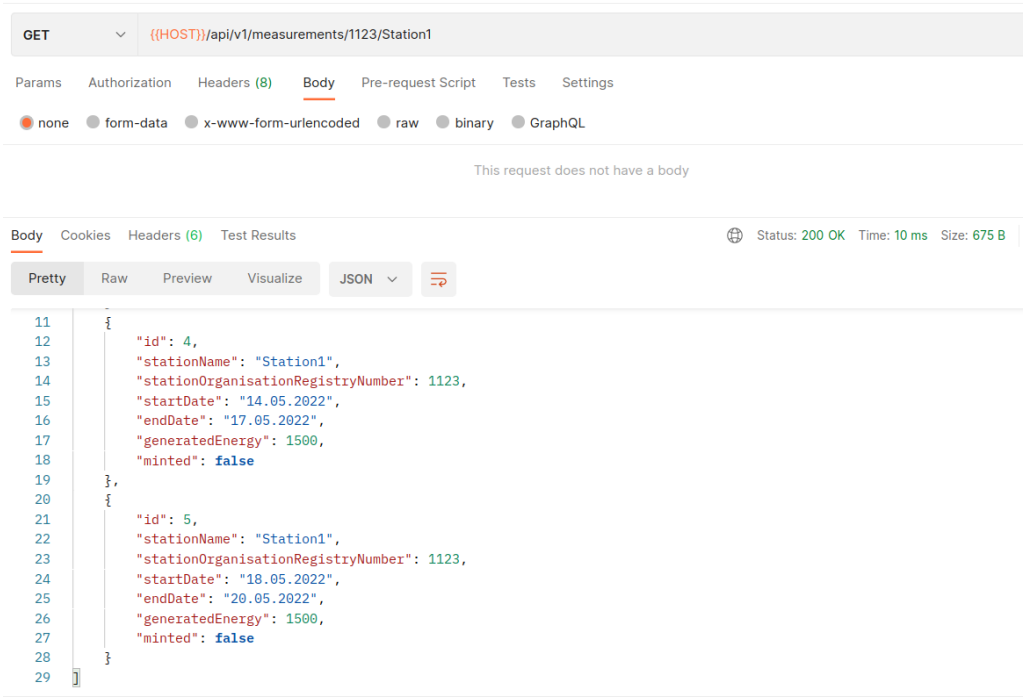


Рис 3.20 - виміри по станції

Випускаємо сертифікат енергетичних атрибутів

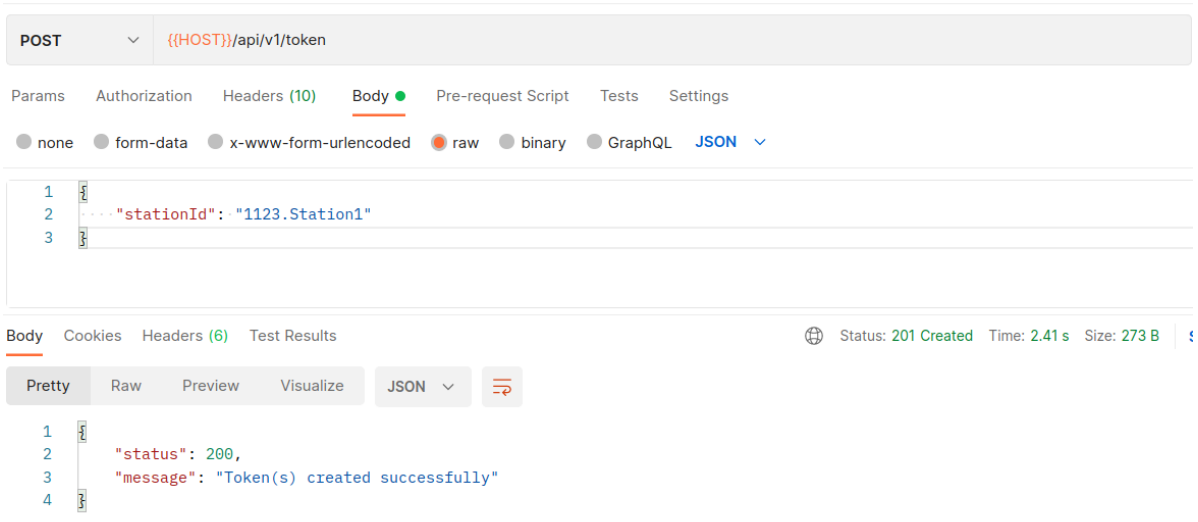


Рис 3.21 - створення токену

Двічі створити його не можливо

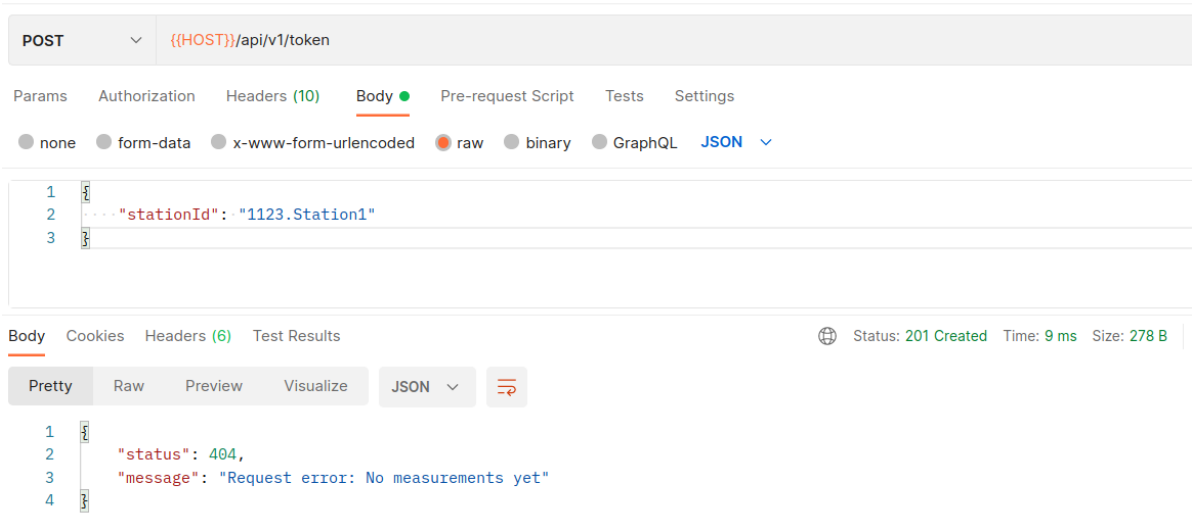


Рис 3.22 - Обробка помилок при створенні токену

Отримуємо щойно створений сертифікат

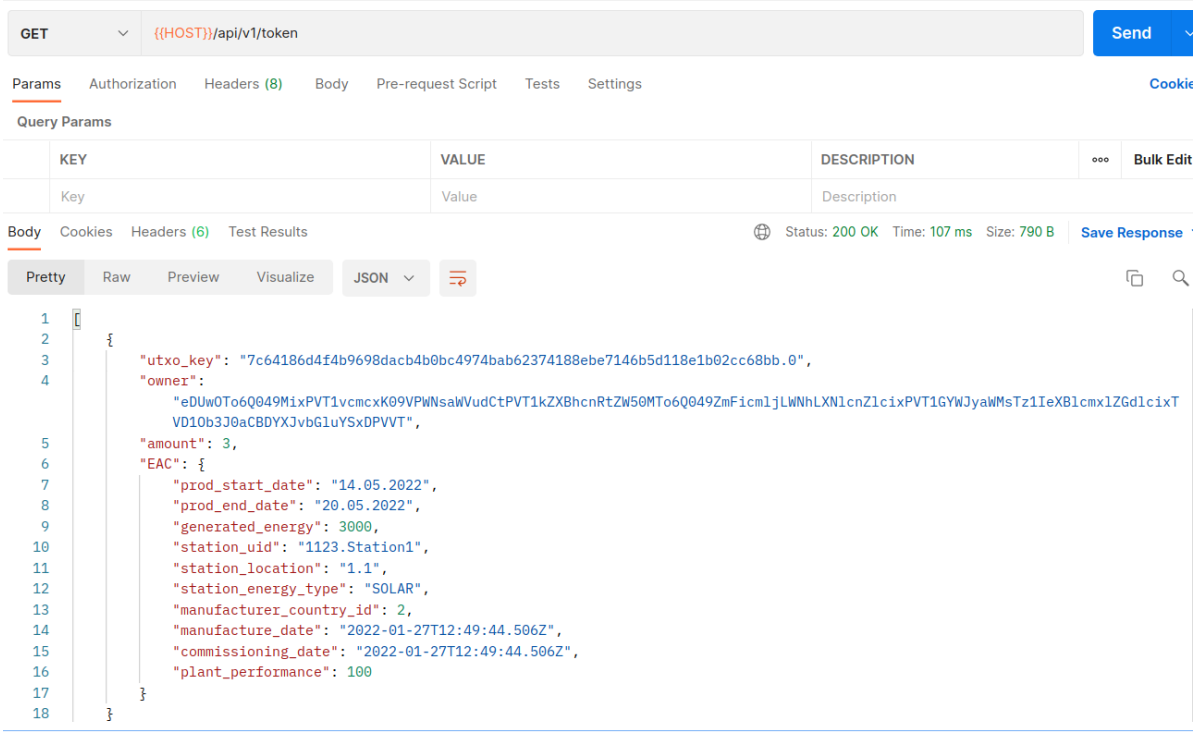


Рис 3.23 - отримання сертифікату

Виставляємо його на продаж

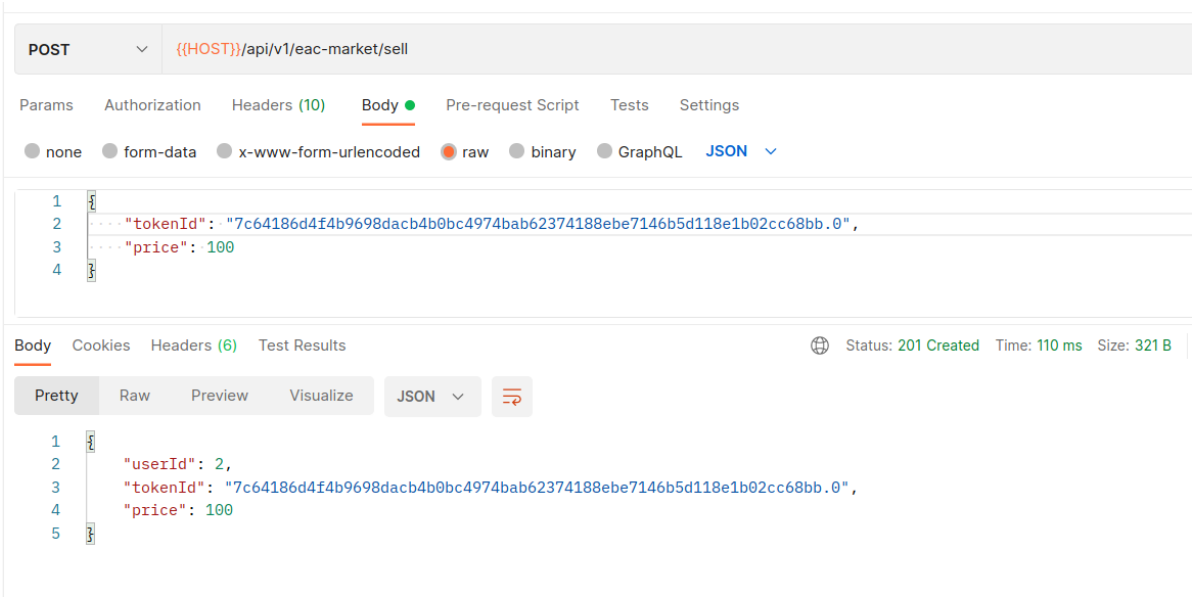


Рис 3.24 - продаж сертифіката

Поповнюємо гаманець

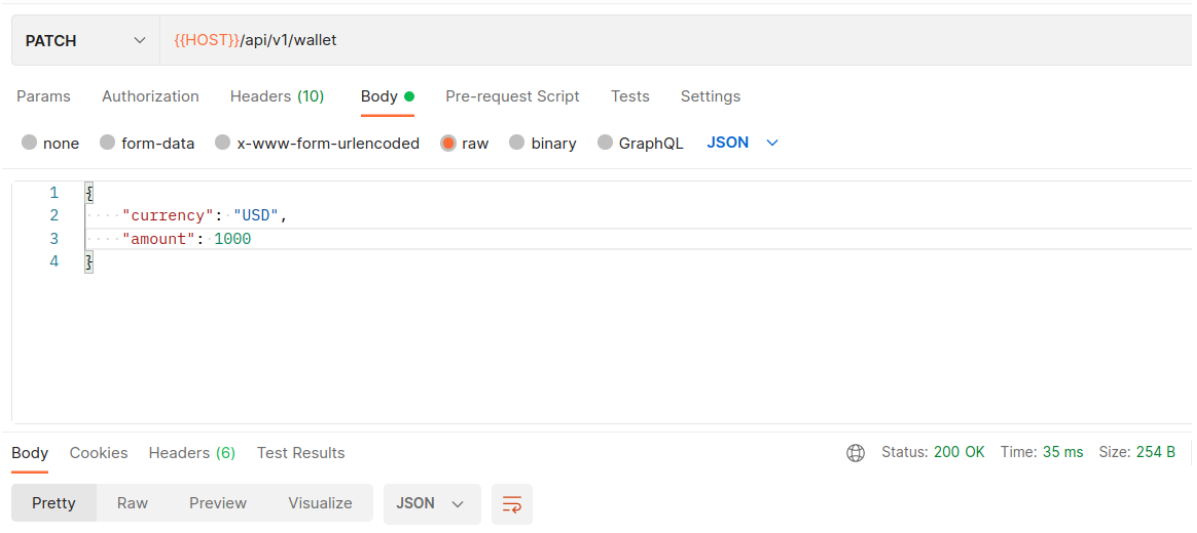


Рис 3.25 - поповнення гаманця

Інвестор передивляється виставлені на продаж EACs

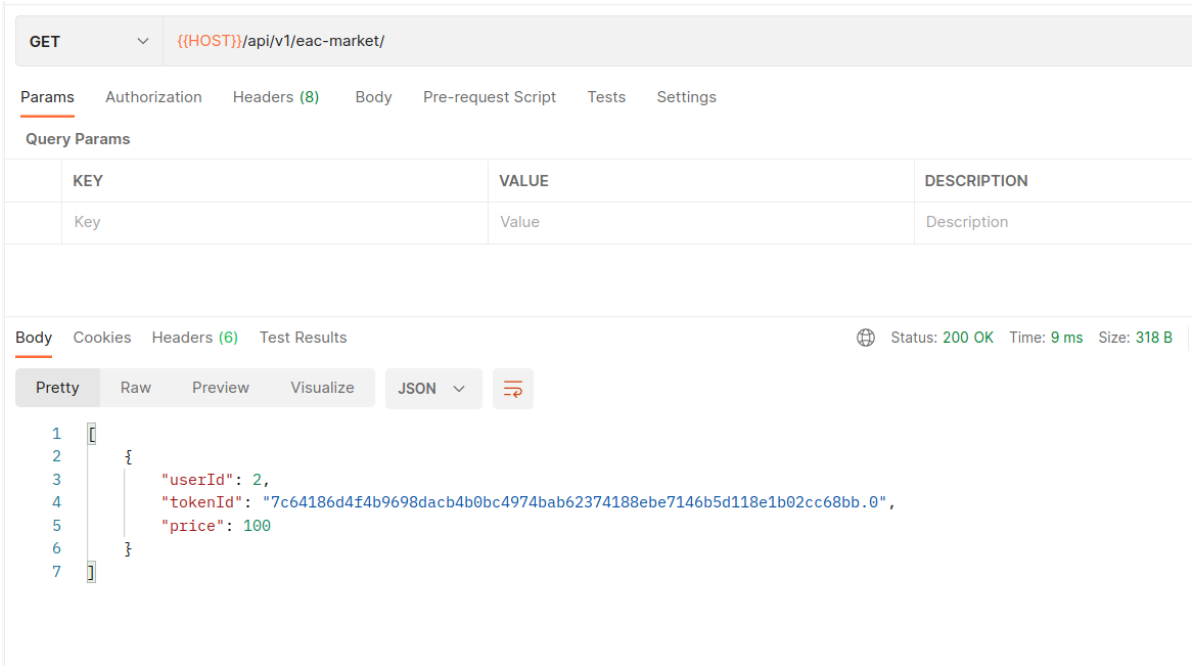


Рис 3.26 - виставлені на продаж EACs

Інвестор купує вказану кількість електроенергії

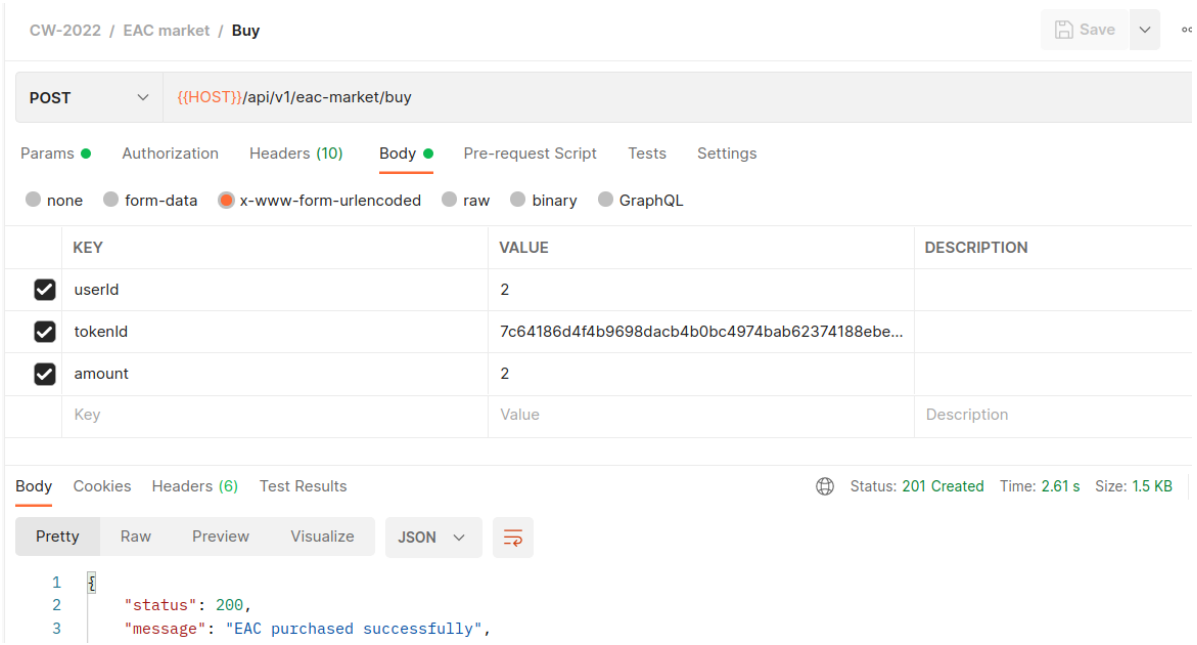


Рис 3.27 - покупка EAC

Отримуємо придбаний сертифікат

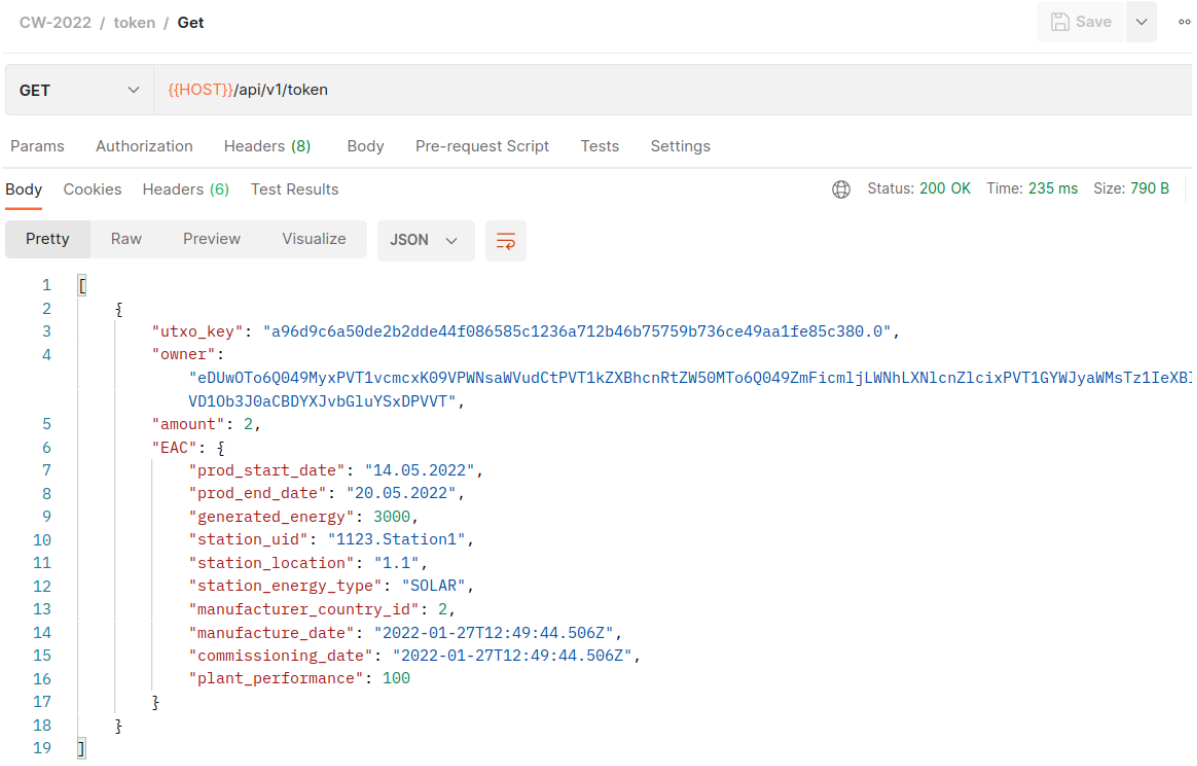


Рис 3.28 - Читання придбаного сертифікату

Фіатний баланс інвестора змінився

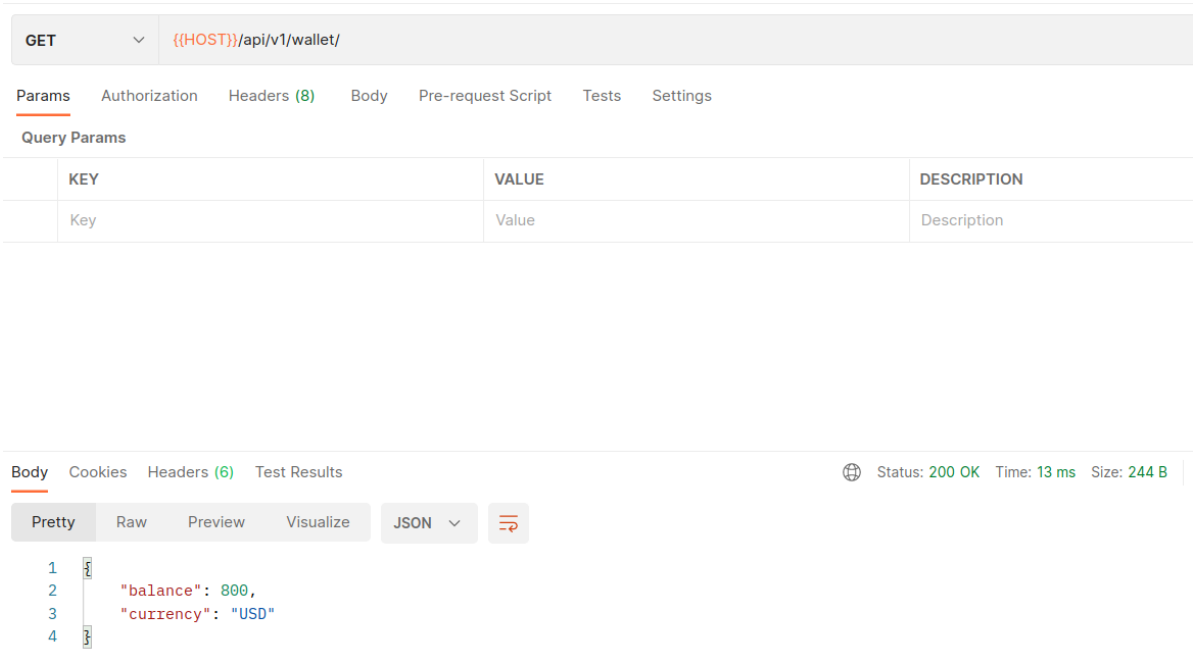


Рис 3.29 - Зміна балансу інвестора

Значення токена у власника компанії змінилися

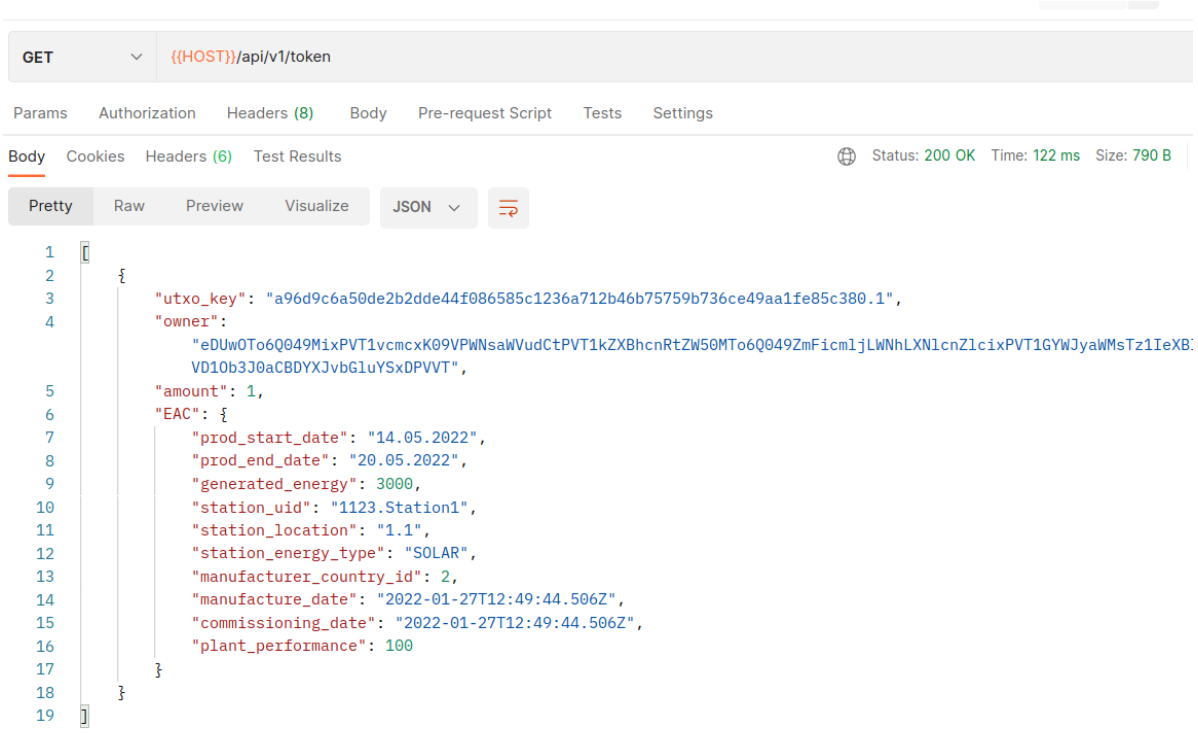


Рис 3.30 - зміна значення токена у власника компанії

Фіатний баланс поповнено

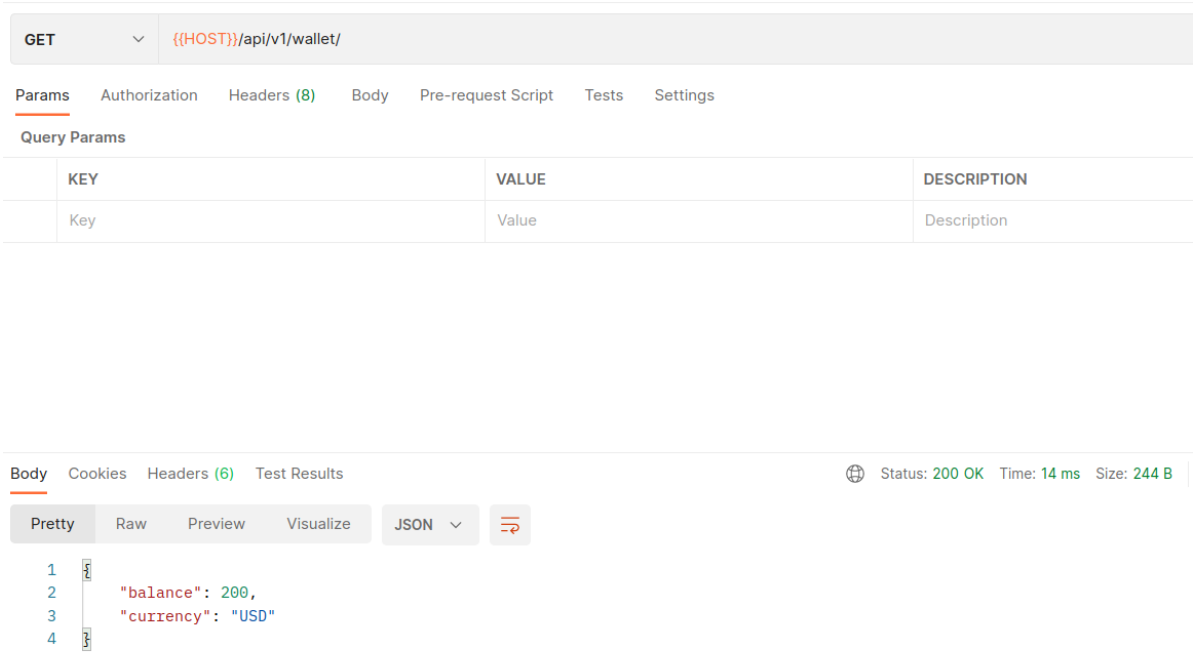


Рис 3.31 - поповнено фіатного балансу власника компанії

Висновки

Отже, поставлене завдання було виконано, а саме: було розроблено програмний застосунок для випуску, продажу та обліку сертифікатів енергетичних атрибутів (EACs), реалізованих за допомогою розширеної UTXO моделі токена. Реалізовано DLT-адаптер для фреймворку Hyperledger Fabric та Fiware Orion Context Broker як спосіб вирішення задачі збору даних про видобуту електроенергію.

Обрані засоби розробки задовільнили технічні вимоги до системи. Розширена UTXO модель дозволила забезпечити фіксацію в блокчейні сертифікатів енергетичних атрибутів, тим самим гарантувати походження чистої електроенергії. Фреймворк Hyperledger Fabric задовільнив потребу в створенні приватної мережі, до якої мають доступ лише інвестори та власники компаній, що мають електронний сертифікат цифрової особи, тим самим забезпечуючи прозорість діяльності клієнтів системи. Контекстний брокер Fiware Orion дозволив ефективно та надійно збирати дані зі станцій.

Створена система є розподіленою мережею вузлів, з якою надається можливість взаємодіяти за допомогою API. Контекстный брокер акумулює інформацію з різних електростанцій та передає її на бекенд, який взаємодіє з блокчейн мережею Fabric. Система має напрямки для подальшого вдосконалення: створення книги замовлень (Order Book), реалізація атомарних транзакцій на бекенді, розширення можливостей та підвищення ефективності системи за допомогою черги повідомлень.

Список використаної літератури

1. United Nations Sustainable Development [Електронний ресурс] – Режим доступу до ресурсу: https://www.un.org/sustainabledevelopment/
2. United Nations Sustainable Development [Електронний ресурс] – Режим доступу до ресурсу: https://www.un.org/sustainabledevelopment/ru/energy/
3. AIB [Електронний ресурс] – Режим доступу до ресурсу: https://www.aib-net.org/
4. Трансформація — 2050: що потрібно енергетичній галузі від діджиталізації [Електронний ресурс] – Режим доступу до ресурсу: https://razumkov.org.ua/statti/transformatsiia-2050-shcho-potribno-energetychnii-galuzi-vid-didzhytalizatsii
5. ТЕОРЕТИЧНІ ОСНОВИ ФОРМУВАННЯ СИСТЕМИ ТОРГІВЛІ «ЗЕЛЕНИМИ» СЕРТИФІКАТАМИ В УКРАЇНІ [Електронний ресурс] – Режим доступу до ресурсу: https://mmi.fem.sumdu.edu.ua/sites/default/files/mmi2017_4_374_383.pdf
6. Blockchain for sustainable energy and climate in the Global South [Електронний ресурс] – Режим доступу до ресурсу: http://www.socialalphafoundation.org/wp-content/uploads/2022/01/saf-blockchain-report-final-2022.pdf
7. Horizen Academy [Електронний ресурс] – Режим доступу до ресурсу: https://academy.horizen.io/technology/expert/utxo-vs-account-model/
8. Fiware [Електронний ресурс] – Режим доступу до ресурсу: https://fiware-tutorials.readthedocs.io/en/stable/iot-agent/index.html

9. Hyperledger Fabric CA [Електронний ресурс] – Режим доступу до ресурсу: https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html?highlight=Certificate%20Authority#certificate-authorities
10. About Fiware [Електронний ресурс] – Режим доступу до ресурсу: https://www.fiware.org/about-us/
11. NestJS Docs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/
12. Fabric Docs [Електронний ресурс] – Режим доступу до ресурсу: https://hyperledger-fabric.readthedocs.io/en/release-2.2/index.html
13. Fabric Test Network [Електронний ресурс] – Режим доступу до ресурсу:https://hyperledger-fabric.readthedocs.io/en/release-2.2/test_network.html?highlight=test-network

Додаток А

Реєстрація цифрової особи

```
async registerAndEnrollUser(caClient, wallet, orgMspId, userId, affiliation) {
    try {
        // Check to see if we've already enrolled the user
        const userIdentity = await wallet.get(userId);
        if (userIdentity) {
            throw {
                status: 401,
                message: `An identity for the user ${userId} already exists in the wallet`
            };
        }

        // Must use an admin to register a new user
        const adminIdentity = await wallet.get(this.ADMIN_USER_ID);
        if (!adminIdentity) {
            throw {
                status: 401,
                message: 'An identity for the admin user does not exist in the wallet. Enroll the admin user before retrying'
            };
        }

        // build a user object for authenticating with the CA
        const provider = wallet.getProviderRegistry().getProvider(adminIdentity.type);
        const adminUser = await provider.getUserContext(adminIdentity, this.ADMIN_USER_ID);
```

```
    // Register the user, enroll the user, and import the new identity into the wallet.
    // if affiliation is specified by client, the affiliation value must be configured in CA
    const secret = await caClient.register({
        affiliation: affiliation,
        enrollmentID: userId,
        role: 'client'
    }, adminUser);
    const enrollment = await caClient.enroll( req: {
        enrollmentID: userId,
        enrollmentSecret: secret
    });
    const x509Identity = {
        credentials: {
            certificate: enrollment.certificate,
            privateKey: enrollment.key.toBytes(),
        },
        mspId: orgMspId,
        type: 'X.509',
    };
    await wallet.put(userId, x509Identity);
```

```
    ,,
    await wallet.put(userId, x509Identity);
    return `Successfully registered and enrolled user ${userId} and imported it into the wallet`
} catch (error) {
    throw error;
}
```

Додаток Б

Придбання EAC

```
public async buyEac(buyerId, tokenOwnerId, tokenId, amount) {
    try {
        // 0. findOne token
        const eacToBuy = await this.eacRepository.findOne( conditions: {userId: tokenOwnerId, tokenId: tokenId});
        if (!eacToBuy) throw new NotFoundException( objectOrError: 'Eac not found');
        // 1. check balance of buyer
        const buyerBalance = await this.walletService.balance(buyerId);
        const token = (await this.tokenService.getClientUTXOs(tokenOwnerId)).find(token => token.utxo_key === tokenId);
        if (!token) throw new NotFoundException( objectOrError: 'Token not found');
        if (token.amount < amount) throw new BadRequestException( objectOrError: 'Purchase amount exceeds token amount');
        const withdrawalAmount = eacToBuy.price * amount;
        if (buyerBalance < withdrawalAmount) throw new BadRequestException( objectOrError: 'Insufficient balance');
        // 2. withdraw fiat
        await this.walletService.update(plainToInstance(UpdateWalletDto, plain: {
            userId: buyerId,
            currency: FiatCurrencyEnum.USD,
            amount: -withdrawalAmount,
        }));
```

```
    // 3. transfer token
    const newTokensArray = await this.tokenService.transferByKeyAndAmount( body: {
        userId: tokenOwnerId,
        recipientId: buyerId,
        token: tokenId,
        amount,
    });
    this.logger.verbose( message: `Token transferred successfully`, newTokensArray);
    // 4. deposit owner's balance
    await this.walletService.update(plainToInstance(UpdateWalletDto, plain: {
        userId: tokenOwnerId,
        currency: FiatCurrencyEnum.USD,
        amount: withdrawalAmount,
    }));
    // 5. update EAC (delete/update)
    if(token.amount === amount) {
        await this.delete(tokenId, tokenOwnerId);
    } else {
        const tokenOwnerPublicKey = await this.userService.clientId(tokenOwnerId);
        const newTokenId = newTokensArray.find(token => token.owner === tokenOwnerPublicKey).utxo_key;
        await this.update(tokenOwnerId, plainToInstance(UpdateEacDto, plain: { oldTokenId: tokenId, tokenId: newTokenId }))
    }
    this.logger.verbose( message: `EAC purchased successfully`);
```